

UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering, Science and Mathematics

A group design project report submitted for the award of
Master of Engineering

Supervisor: dr monica mc schraefel
Examiner: Dr Tim Chown

mSpace: Exploring The Semantic Web

by Craig Harris, Alisdair Owens, Alistair Russell
and Daniel Alexander Smith,

December 15, 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

A group design project report submitted for the award of Master of Engineering

by Craig Harris, Alisdair Owens, Alistair Russell and Daniel Alexander Smith,

Information on the web is traditionally accessed through keyword searching. This method is powerful in the hands of a user that is experienced in the domain they wish to acquire knowledge within. Domain exploration is a more difficult task in the current environment for a user who does not precisely understand the information they are seeking. Semantic Web technologies can be used to represent a complex information space, allowing the exploration of data through more powerful methods than text search. Ontologies and RDF data can be used to represent rich domains, but can have a high barrier to entry in terms of application or data creation cost.

The mSpace interaction model describes a method of easily representing meaningful slices through these multidimensional spaces. This paper describes the design and creation of a system that implements the mSpace interaction model in a fashion that allows it to be applied across almost any set of RDF data with minimal reconfiguration. The system has no requirement for ontological support, but can make use of it if available. This allows the visualisation of existing non-semantic data with minimal cost, without sacrificing the ability to utilise the power that semantically-enabled data can provide.

Contents

1	Why We Need mSpace	7
1.1	Finding Information From The Web	7
1.2	What Is An mSpace?	7
1.3	Preview Cues	8
2	Background	9
2.1	The Semantic Web	9
2.2	Semantic Web Technologies	9
2.3	Using RDF Data	9
2.4	TripleStore	10
2.5	3Store on OS X	10
2.6	RDF Data Query Language	11
2.7	Semantic Web Tools	11
3	User Interface	13
3.1	Interaction Issues	13
3.2	Power Assist	13
3.3	Instrumentation	14
3.4	Information Box	14
3.5	Goal Counting	15
4	Data	16
4.1	Classical Music Domain	16
4.1.1	Piece Information	16
4.1.2	Recording Information	16
4.1.3	Data for Information Box	17
4.2	IMDB Domain	17

4.2.1	Movie Data	18
4.2.2	Trailers	18
5	Design	20
5.1	Introduction to the mSpace UI	20
5.2	Functions of the UI	20
5.3	Implementation Tools & Language	22
5.3.1	Server Side or Client Side?	22
5.3.2	Flash, Java or JavaScript?	22
5.4	User Interface Design	23
5.4.1	General Layout	23
5.4.2	Column Layout	23
5.4.3	Column Design	24
5.4.4	Information Box Design	25
5.4.5	Favourite List Design	25
5.4.6	Preview Cue Design	26
5.4.7	Column Menu Design	27
5.4.8	Status Bar	28
5.5	Generalisation	29
5.5.1	Ontological Model For mSpace	29
5.5.2	UI Generalisation Theory	31
5.5.3	UI Generalisation Implementation	31
5.5.4	IMDB	32
5.5.5	Relevance of Previews	32
6	User Interface Implementation	34
6.1	Implementation	34
6.1.1	Simple Object Class Diagram	34

6.1.2	JavaScript Objects	35
6.1.3	HTML Document Object Model	35
6.1.4	Column Object	36
6.1.5	Column Container Object	38
6.1.6	Favourites List Object	39
6.1.7	Information Box	41
6.1.8	List Object	42
6.1.9	List Item Object	43
6.1.10	Menu Item Object	45
6.1.11	Popup Menu Object	46
6.1.12	Preview Cue Object	47
6.1.13	Preview Item Object	49
6.1.14	Preview List Item Object	49
6.1.15	Removable Preview List Item Object	51
6.1.16	Status Bar Object	51
6.1.17	Generic Functions & Variables	52
6.2	User Interface Event Chain	54
6.2.1	User Interface Bugs & Features	55
7	Controller Implementation	58
7.1	Data Structures	58
7.2	Representing a Displayable Item	58
7.3	Column Information	58
7.4	Populating the Column Array	60
7.4.1	Multi-hop data	63
7.5	Performing queries - XMLHttpRequest	63
7.6	Preview Cues	64

7.7	Favourites	65
7.8	Power Assist	65
7.8.1	Passive Power Assist	65
7.8.2	Active Power Assist	66
7.9	Communicating with the User Interface	66
7.9.1	Preview Cues	66
7.9.2	Favourites	67
7.9.3	User initiated events	67
7.9.4	Updating the Screen	68
7.10	Generalisation of Controller	68
8	Other Implementation	71
8.1	SourceForge	71
8.2	Populating the Information Box	71
8.3	Instrumentation	73
8.4	Goal Count Methodology & Scalability	74
9	Testing	75
9.1	Incremental Delivery	75
9.2	User Acceptance Tests	75
9.2.1	Useful Suggestions	75
9.2.2	Era Sort	75
9.3	IMDB - Generalisation test	76
9.4	Test Plans	76
10	Evaluation	77
10.1	Time Management	77
10.2	User Interface Future Work	79

10.2.1	UI Component Library	79
10.2.2	Further UI Generalisation	79
10.2.3	Column/List Filter	79
10.3	Future Work	79
10.3.1	Scalability	79
10.3.2	Improving parsing performance	80
10.3.3	Improving query times	80
10.3.4	Other performance enhancements	80
10.3.5	Streaming	81
10.3.6	Multiple selections in a column	81
10.3.7	Admin tools	81
10.3.8	Instrumentation Analysis	82
A	Information Box Code Sample	84
B	3Store On OSX - HOWTO	85
C	User Acceptance Tests (UATs)	87
D	Annotated Instrumentation Log Entry	90
E	List Of Test Criteria	92

1 Why We Need mSpace

1.1 Finding Information From The Web

Access to information on the web has traditionally been through the use of keyword searching[6]. This is a very powerful way to access information, given that the user has domain knowledge. When a user's knowledge of a particular domain is limited, there is no currently existing mechanism by which they can learn (through the web) about a domain to which they have limited or no knowledge. The effect of this is that a user with domain knowledge will add domain-specific constraints to a search that a domain-naive user will not be able to. mSpace can be used to allow the domain-naive user to explore the domain, learning about it themselves.

The example we discuss is a user that does not know much about classical music, but when they have been exposed to it before, have liked some of what they have heard. The aim of the user is to ultimately find music that they like, and to understand which aspect of the music they like. They may be partial to the works of a particular composer, an entire era, or perhaps they just really like string arrangements.

A typical approach would be to load google and submit a search of "classical music". Google will faithfully return a listing of all pages that contain the words "classical" and "music". In order to learn about some specifics on classical music, the user would need to add extra constraints, which they do not yet know.

1.2 What Is An mSpace?

The main feature of an mSpace is to make the exploration context-sensitive, in the user has a goal in mind and that their browsing should retain that goal as an outcome at all times. In order to do this, multiple dimensions are created within the space (the word mSpace is an abbreviated form of "multi-dimensional space"). A collection of dimensions within a space are known as a slice, and it is often useful to imagine the graphed data being sliced through with an mSpace. The user can make selections from these dimensions which influence their resulting goal items.

In the case of classical music, there are dimensions such as "Composer", "Era" and "Arrangement". The mSpace interface allows the user to make selections from these dimensions to find pieces of music which they like. There needs to be an indicator of the differences between each selection, and why they are of interest to the user. In the case of classical music, it would be appropriate to allow the user to hear a relevant sample of the music that is associated with the selections, so that the user can decide which particular route to follow through the mSpace. These samples are known as "preview cues" [35].

1.3 Preview Cues

A preview cue allows the user to experience relevant samples of a category via a lightweight interaction. In the case of exploring an *mSpace*, this is useful when the “goal” dimension of the space is multimedia, for example, when exploring a music space, a video space or image space. Theoretically this could also be applied to other areas through thumbnailing (i.e. websites)[37] or simply providing non-multimedia content, such as an excerpt from a book.

2 Background

2.1 The Semantic Web

The semantic web aims to extend the current web from its current human-readable state into a machine-understandable one[25], in order to gain more value from the effort and data that is being put in by so many people[22]. Through the W3C[23] the standards base for the semantic web is being formalised[30]. While some may not agree[38] with the W3C's choice of technologies (see Section 2.2), the general ideas, are living on in initiatives such as the friend of a friend (FOAF) programme[5], which has seen a rise in popularity in the recent surge of blogging[34] and the use of so called Really Simple Syndication (RSS) feeds[15].

Users of such systems as FOAF and RSS are reaping the value of semantically-enabled data on a daily basis. RSS is used to notify updates to a site (normally a blog or a news site, both of which can be expected to have many updates per day), through a desktop aggregator, and increasingly through mechanisms built into the browser[28]. Version 1.0 of RSS is in fact fully RDF compliant and in use by many people, without them even knowing anything about the semantic web. FOAF is used on blogs, personal homepages and social networking sites (such as orkut[12] and tribe[19]). It presents an ontology[4] to describe yourself and link to people you know. The popularity has risen since a tool called FOAF-a-matic[2] was created and publicised that allows people to easily create a "FOAF file" to place on their websites. There is a great deal of power available in these as it allows a crawling system to create a network of users with links to who they know[3], and has really created the kind of value described above.

2.2 Semantic Web Technologies

There are two key technologies that are used in the semantic web. The Resource Description Framework (RDF)[16] is used to define data as a graph. Entities are created that represent the nodes of the graph and are linked using predicates. In order to add semantic significance to the nodes and predicates, the Web Ontology Language (OWL)[21] is used to specify nodes in terms of defined classes and inheritance is used to infer more about the graph properties.

2.3 Using RDF Data

When dealing with RDF data (or indeed, any graphed dataset) it is important to be able to quickly and efficiently query the knowledge base. In traditional systems, flat files were used, and system calls were quickly made more efficient at reading from files and being able to allow very fast random access to that data[26]. Relational database

systems built upon this flat file efficiency to allow data to be normalised[31] and hooked together, whilst still allowing efficient querying, normally through the use of the Structured Query Language (SQL)[10]. While this data was graphed to a certain extent, it did not possess the ad-hoc definition and inference through inheritance that can be achieved using RDF[16]. It would be foolhardy to now return to flat files and ignore the technological innovations in speed and scalability of modern database systems. As such, mSpace interacts with the data through software known as a triplestore.

2.4 TripleStore

The basic element of graphed data is the triple[14]. The relationship of a subject, through a predicate to a resultant object can be represented as a list of three items, known as a triple. As the name suggests, storing these with the intention of allowing triple-based queries is the job of a triplestore.

Implementation methods of triplestores vary depending on platform, data storage method and optimisation technique. For example, 3store[1] is optimised for URI-oriented data[32], whereas IFPStore[29] is optimised for inverse functional property (IFP)[13] oriented data. mSpace queries 3store, a triplestore that uses the MySQL relational database server software as storage and acts as a layer between the raw triples in the database and the RDF Data Query Language (RDQL) (see Section 2.6).

Given the generic nature in which mSpace implements querying, it is reasonable to assume that with few changes mSpace could query different triplestore software. RDQL is an open standard[36] and as such, mSpace is inherently not tied to a proprietary server solution and is extremely flexible in how the data can be supported.

In the case of 3store (and indeed most triplestore software), the data is asserted into the knowledge base by importing RDF files. This data is then parsed by the triplestore and all relations are created inside the knowledge base between the new data and the existing data and reflected in subsequent queries to the knowledge base.

2.5 3Store on OS X

For development of the mSpace project, a great resource was made available to the team in the form of an Apple XServe server. This computer was extremely useful for the large-scale of imports of data that were performed during development.

One issue that resulted from using this server was with the 3store software, which was untested on the OS X platform which the XServe runs[27]. Several issues with the code that prevented it from compiling under the OS X platform were resolved, the patches sent to the 3store team and integrated into the 3store software. A document that explains how to install 3store under OS X was also created (See Appendix B).

2.6 RDF Data Query Language

The RDF Data Query Language (RDQL) is an SQL-like query language used to extract data from RDF graphs. An RDQL query is a graph pattern expressed as a list of triple patterns which can have constraints on the values of variables listed in the pattern, as well as control over which variables are returned[36]. While it is much more basic than SQL, it provides the functionality needed to extract the data we require. An example query is shown in Figure 1:

```
SELECT ?y WHERE (?x, <mspace:has-era>, ?y) USING mspace FOR
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/>
```

FIGURE 1:

In this query, the variable *x* is the subject, `<mspace:has-era>` is the predicate, and *y* is the object. Note that the question mark denotes that the item is a variable. The query finds all *x* that have an era *y*; if we take an example of the classical music data that this query applies to, it will find all eras *y* that every piece of music *x* belongs to. We are only SELECTing *y*, so only the URIs of the eras will be returned, not the pieces as well. The USING statement allows the replacement of text with another piece of text for convenience value. In this case the actual predicate would be `<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/has-era>`, but the USING statement allows us to replace that with the neater `<mspace:has-era>`.

```
SELECT ?x WHERE (?x, <mspace:has-era>, <mspace:Baroque-Era>) USING mspace
FOR <http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/>
```

FIGURE 2:

The second example in Figure 2 shows the variable *y* replaced with a URI. Note that in this query we are selecting *x*, or pieces of music. This constrains the query to finding all pieces of music that have the Baroque era.

2.7 Semantic Web Tools

At this time there is quite a high barrier to entry in producing semantic web sites and as such it is in the interest of everybody that it be made easier to semantically-enable existing web presences as well as create new web presences that are semantically-enabled from their inception. One of the goals of this project is to create a generalised system that allows for a very low cost entry into the semantic web so that the benefits can be seen and experienced by users and developers who are naive or sceptical about the semantic web and any benefits they can expect from it.

It is unreasonable to expect an enterprise to convert their entire web presence or back-end systems to a semantic one overnight, indeed, without new applications to utilise

the semantics it is unlikely that value would be realised of such an action. As such, it is necessary to demonstrate the viability of the semantic web through the creation of systems that allow low-cost production of feature-rich data-based applications that would take more cost and time to produce using traditional non-semantic data.

mSpace is such a system and demonstrates that with only a small file of definitions it is possible to create an extremely flexible, yet constrainable system. To lower the cost further, a GUI tool to create this file of definitions could be created, although that is outside the goals of this particular project. It would also be fairly straightforward to create a GUI tool to convert existing data into data graphs suitable for use with mSpace. Although they would have little semantic value, many of the benefits of the semantic web, in that a system can be created cheaply, would still be present and demonstrable.

3 User Interface

3.1 Interaction Issues

Human Computer Interaction (HCI) is the study of the ways in which humans interact with computers and includes the evaluation of the methods involved. mSpace is concerned with providing users with a simple and lightweight interface for the exploration of multidimensional data and many HCI issues need to be considered during the development of the solution. Any interaction techniques which could confuse or frustrate the user need to be avoided along with any inefficient operations. It is sometimes beneficial to overlook the difficult nature of some interaction methods in favour of the familiarity that end users are already likely to have regarding the particular method. Addressing HCI issues improves the usability of products; when considered on its own usability is regarded as the effectiveness, simplicity and enjoyment of interaction, from the perspective of the user.

The web-based nature of the user interface could make users think twice about clicking on elements they are interested in, because much of the internet is slow and users frequently waste time waiting for pages to load. In most cases it is infeasible to change such interaction techniques, but where possible events in mSpace are triggered by moving the mouse over a particular target area, e.g. a preview button. The downside of firing events due to mouse movement is the risk of unintentional actions when a user merely wants to move a mouse past a target area. This can be counteracted by means of small delays, we allow a mouse click as well as mouse-over thus meaning that the user can instantly interact with the interface, bypassing the delay. The surprise of a mouse rollover event may be slightly annoying to some users, but it also generates further interest in the interface and is quite easy to learn; memorability is also important and consistency within a user interface helps to ensure such. The combination of techniques mentioned above is implemented in mSpace to best maximise the users' experience.

3.2 Power Assist

One of the issues that this project brings to light is that users do not always browse for data in an efficient manner. While the mSpace column layout aids the user in discovering what types of music they enjoy, it is entirely possible that the user will not notice some trends in their selections. The idea behind a Power Assist function is to make these trends known to the user to enrich their browsing experience.

To accomplish this, we identified two methods that could be of use. Firstly, an active function that rearranges column layouts and selections at the request of the user. An example of the use of this can be found in a user that is browsing through every different composer and selecting tracks that are arranged for a particular instrument. This active

assistance might offer to change the columns to filter everything first by instrument played. This, as well as helping the user to browse in a more efficient manner, could also help the user to identify what makes them enjoy particular pieces. It should be noted that a step such as changing user-specified layout and selections should only take place if explicitly requested. Automatic functions that undertake such actions are usually considered an annoyance.

The secondary form of Power Assist is more passive. This involves highlighting items within the existing layout that may be of interest. Thus, when a user makes selections or opens a new column, they may notice that their selections have a common factor within the new data.

3.3 Instrumentation

Many user interfaces include features which seem like a good idea when conceived, but which may not actually be useful once implemented and deployed. The best way to measure the usefulness of aspects of a user interface is to monitor the ways in which users interact with them. Supervising or video recording the use of a prototype can highlight some short fallings, but is an intensive task. A more efficient way of monitoring the ways in which users interact with a system is by automatically logging everything they do and then apply statistical analysis to the logged data; this process is known as instrumentation.

Instrumentation includes a way of storing metadata regarding events a user caused and participated in; the data is generated and collected by augmenting additional function calls into the user interface module. The instrumentation data can be used to prove the effectiveness of each part of the user interface. When things go wrong, instrumentation data can also be used to recreate scenarios which led to failures. This may somewhat assist the debugging process.

3.4 Information Box

Many users of an mSpace interface will have little knowledge of the domain being explored. It is of great benefit to provide additional knowledge to the user so that they may make more informed decisions. The provision of an information area on the screen allows additional knowledge to be conveyed to the user in a non-intrusive manner and makes exploration of an mSpace a more educational experience.

3.5 Goal Counting

The data in slices of a space could potentially have billions of permutations; there could be hundreds of thousands of goal instances and a choice of many columns with which to narrow down the goal list to a precise collection. The spread of goal items across such columns is likely to be uneven. When searches are executed on traditional search engines, the user is informed of the number of matches, from which the user can deduce the suitability of the search term(s); similar information can be augmented within column listings so that users can quickly attain a digestible quantity of relevant goal items.

The complex relationships between the dimensions of knowledge coupled with the sheer volume of data potentially present in a knowledgebase along with the limitations of RDQL means that the computation of goal counts is far from a simple efficient task; such will be further discussed in Section 8.4.

4 Data

4.1 Classical Music Domain

4.1.1 Piece Information

When deciding on the correct ontologies with which to define data and metadata it is important to ensure the correct semantics are applied. This allows for the greatest quality of the data, with derived value through inference. In order to decide on which ontologies to use, one must first know what ontologies have been created. Searching the database at SchemaWeb[17] resulted in a match with a classical music ontology[33]. The semantics of this ontology are appropriate and of a high quality, although not all of the data that we have in this domain can be expressed, so we have added some of our own predicates and classes to fully utilise the data.

Use of ontologies and correct semantics is of use to the data and useful for advanced inference through the 3store software (see Section 2.4). mSpace itself does not see this and sits atop the data and semantics, interacting with data as an output, because of this it is important to realise that if an mSpace was to be created for a space which does not have semantic data, that ontologies are not required.

In the spirit of this, given the time constraints of the project[20] and that creation of a semantic classical music space is not the main focus of this project, we've not explored the full semantic correctness of the new predicates that were applied. This has no effect on the exploration of the space. Indeed, a good use (in terms of mSpace) of ontologies and more specifically OWL[21], would be to perform ontology mapping[11] from existing data to the predicates required for certain elements of the mSpaceModel ontology (see Section 5.5.1), such as preview cue links.

4.1.2 Recording Information

Once a user has decided upon the pieces they enjoy listening to, they need to know which recordings exist of them, and which albums those recordings appear on. Once they know the album name it is easy for them to locate and purchase copies of the album. Each recording will typically have been done by a different modern day artist, even though the piece could have been composed hundreds of years ago; each artist will contribute in a slightly different way to a recording and therefore users should be able to get the list of tracks by a particular artist.

4.1.3 Data for Information Box

There are two main ways in which users can learn about a particular domain using an mSpace. Firstly they can explore the data and realise the relations and any themes across the dimensions; secondly they can read specific information about any item presented within the slice. There is a broad range of information which can potentially be presented to the user as a description of a selected item, an example of which would be to display the biography of a composer. It may also be desirable to display a picture of the composer and separately give information about which types of music they composed and when they composed them.

ClassicFM.com has lengthy descriptions of each era and also has biographies of many of the more popular composers. The information available on ClassicFM.com is of benefit to users of a classical music mSpace, and thus it was extracted and stored including copyright information in the TripleStore. The extraction of data was accomplished by using a PHP script to parse each information page, searching for known strings which occur before and after the actual data. The list of pages from which to extract information was first obtained by parsing the relevant index pages from the ClassicFM.com website.

The HTML extracted from the information pages was manipulated to become XML compliant which included removing some invalid tags, correcting other tags and encoding special characters. Eras were easily mapped to existing knowledge as the resource identifiers for eras all end with the era name; the composers were slightly trickier as it was necessary to run selection queries on the composer name to find the relevant resource identifier. This was hindered by the varying names used; middle names are regularly missed out and there are alternative names in addition to variations on the order of names and sporadic use of initials.

4.2 IMDB Domain

It was decided that in order to test the generalisation of the project, that another domain should also be described using the mSpace model (see Section 5.5.1). The Internet Movie Database (IMDB) is a domain to which we can link to trailers as preview cues, and to which the information is available in plain text from the source.

The data was parsed and output into RDF from the plain text and asserted into the triplestore. Given the sheer size of the data, this was partitioned into a separate file for each slice, i.e. Genre, Producer, Director are all different files. Even when this partitioning was in use, each file was over 80MB in size.

4.2.1 Movie Data

Given that the data was only to be used as a quick test domain, little time was given to proper use of ontologies, however further research since the creation of the data has led to the discovery of a well-designed ontology for the imdb[24]. For any future use of the imdb data, it is recommended that the data utilise this ontology, for good semantic integrity.

4.2.2 Trailers

Trailer links were not available from the IMDB, and given their policy on crawling the IMDB web site[7], another approach had to be taken. Another web site called The Movie Insider[8] was crawled for approximately 2000 trailer links and matched with the films through the use of some small perl scripts. This gave the bulk of the trailers, and some other sources were also crawled, albeit with a more manual approach, in order to fill in some gaps, particularly for classic movies trailers for films such as “Casablanca” or “The Maltese Falcon” as in Figure 3.

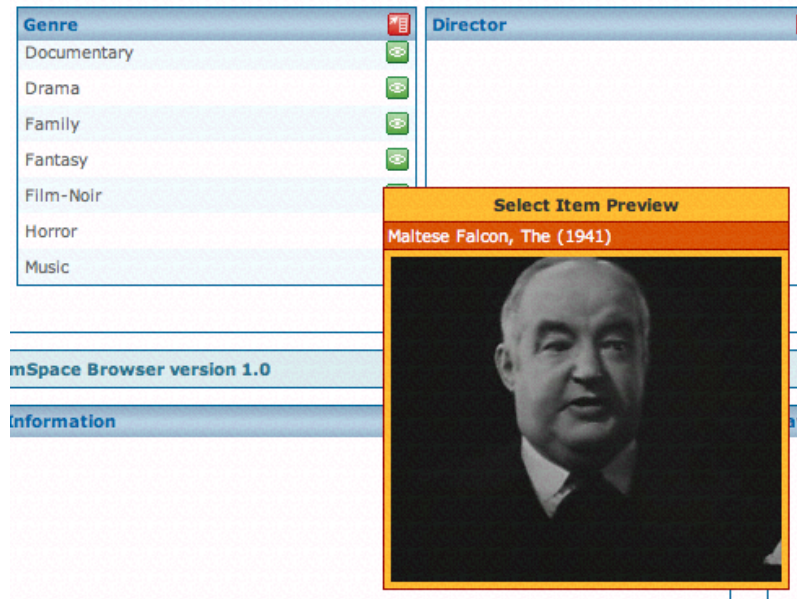


FIGURE 3: Film Noir Preview - “The Maltese Falcon”

In order to minimise delay to the user (which is typical when exploring such a large dataset in this manner), restrictions in the ordering of the dimensions should be enforced, so that no column can ever become populated with an overwhelming amount of results. Alternately, new dimensions could be inferred to enable to partitioning of the data, i.e. in order to partition by year, it may be beneficial to initially have a dimension of “decade”, with the individual years being a subsequent column. These suggestions would become obsolete if some sort of column partitioning system were developed in to

the mSpace project, i.e. an alphabetical auto-sized partition of letters, as in the artists rendering in Figure 4.



FIGURE 4: Conceptual Drawing of Tabbed Column

5 Design

The User Interface is the most important part of the mSpace project. It is the part of the project that actually provides the user with the ability to browse the domain that they are looking at. It needs to provide all the functions that will aid the user in their browsing of the domain, but at the same time it also needs to be a fairly simple interface so that it does not distract the user from the task at hand.

The following sections outline the functions that the mSpace User Interface provides, and looks through the HCI (Human Computer Interaction) issues encountered.

5.1 Introduction to the mSpace UI

The main task of the mSpace project is to allow a user to browse a domain of information that they have no prior knowledge of. The User Interface is used to give them a starting point, and provide help at every stage during their exploration of this information.

If we teach the user about the domain they are exploring, they will learn more. They will be able to use this knowledge to explore the information in a much more informed way. Being able to understand the underlying semantic data, and the relationships between this data, is much more important than just presenting the information to the user. The mSpace UI needs to not only present the domain of information to the user, but also emphasize the relationships between the data so that they can better understand it.

The domain being used for this project is Classical Music. The end objective of the domain exploration or Goal of classical music is a particular piece or a recording of that piece. For example Beethovens 5th Symphony could be a piece the user may not specifically be looking for, but may actually like.

There is a wealth of information associated with each piece, for example, the 5th Symphony was an Orchestral arrangement piece, that was composed by an Austrian composer Ludwig van Beethoven in the Classic era. The mSpace UI should allow the user to explore these relationships and draw conclusions.

5.2 Functions of the UI

The main function of the mSpace UI is to show the data available in a domain, in easily readable columns. In the classical music case, there should be a column that represents each of the categories of information available about a piece.

- Era
- Composer
 - Composer Country
 - Composer Death Date
- Arrangement
- Piece
 - Recording

The previous example categories are not all that might be associated with classical music, and there are many more that could be included. Each of the categories is also not necessarily directly associated with a piece of music, for example the Composer Country is linked to the Composer of a Piece, not the actual Piece itself.

The columns need to allow the user to rearrange them, in that they must provide the ability to add a new column, subtract a column or move columns about. In the Classical Music case, each column just needs to show a list of items, each with a descriptive label and a Preview Cue.

When you click on any item within a Column, you should be able to see relevant information about the selection. Selecting Ludwig van Beethoven in the Composer column should give the user a biography about the composer. An area of the UI can be designated as the information area, which will display this biography (or whatever information is associated) for the user to read, without having to navigate away from the page.

Another function of mSpace is a Favourites List which will allow a user to save a particular goal item, a Piece in the Classical music case, for reference at a later date. The Favourites List should provide a list of items that the user has added. These items need to be stateful so that they remain in the favourites list between sessions.

The following is a breakdown of the required UI functions based on the mSpace model. There are also other functions that need to be implemented that will aid the user in their browsing experience.

- Display columns that contain a list of items
 - The items in the list should have a Preview Cue
- Allow the user to rearrange the columns
 - Add a Column that is not present
 - Subtract a Column that is present

- Move a Column left or right
- Display information about the current selection within a Column
- Have a separate favourites Column
 - Allows items to be added
 - Allows items to be removed
- Have a status bar to display messages to the user

5.3 Implementation Tools & Language

The mSpace project is about exploring the Semantic Web, and it therefore makes sense to create an application that is available through a web browser. This has several advantages such as cross-platform compatibility and also world wide accessibility.

5.3.1 Server Side or Client Side?

Server side languages such as PHP, JSP and ASP all have their merits, but they also have one big disadvantage which is that the page will need to be reloaded every time the user clicks an item. For this reason a client side solution has been chosen.

5.3.2 Flash, Java or JavaScript?

The three main choices when it comes to creating a client side web application are Flash, Java or JavaScript. Flash is a very useful tool for creating dynamic and custom User Interfaces, however the Flash Player it is not an plug-in that everyone on the internet has, and using the Flash development tool goes against the open source nature of this project, since it has an associated cost.

Java is a very powerful language, is used throughout the world and can be developed in for free. Again one of the main drawbacks when creating a Java Applet is that not everyone on the internet will have the very latest version, and along with flash there are other accessibility problems to overcome, with regards to applications such as a Screen Reader.

JavaScript is the obvious choice. It is supported by all the modern browsers and in some form by many of the old browsers. It is free to develop in, and provides less accessibility problems with applications such as Screen Readers than Flash or Java Applets do.

5.4 User Interface Design

Before the UI can be coded some designs need to be formulated, to get an idea of the layout of the web page. The following sub-sections cover the design of each of the elements within the UI.

5.4.1 General Layout

The general layout of the page needs to be simple. The user needs to be able to access all the functions of the UI, and also see all the information on one screen without having to scroll the whole page.

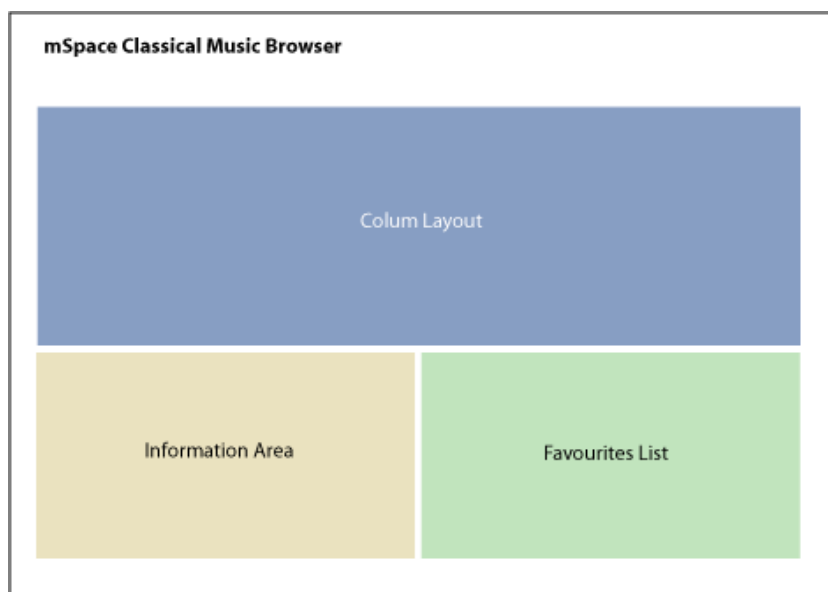


FIGURE 5: Initial Design of the mSpace Browser

Figure 5 shows 4 distinct sections, which will be looked at in more detail, and their relationship to each other on the page.

The Column area, the largest area on the page, is where most of the interaction with the user will occur, where as the Information area and Favourites list equally occupy a much smaller area than the Columns.

The size of the Information area may need to be larger than the Favourites List as the data contained within will potentially be larger than that of the Favourite List.

5.4.2 Column Layout

The Column area may contain any number of columns. Since the Columns are to be navigated from left to right, it makes sense to enable the area to horizontally scroll the

columns if there are too many to fit on the screen. Another option would be to have the columns wrap around on to a new line on the page, but this might push the Information area and Favourites List down the page and out of view. This needs to be avoided.

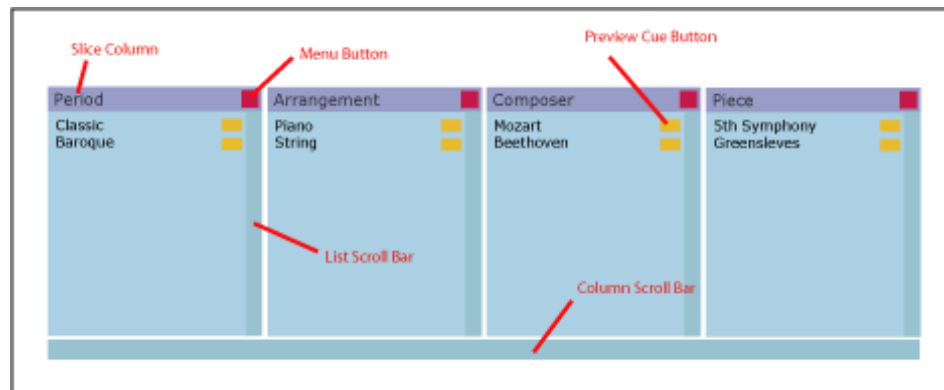


FIGURE 6: Simple Design For The Column Area

As Figure 6 shows, each column is a list with a title and a Menu button, and each item in the list has a Preview Cue button. The Column is designed in more detail in the next section. The Scrollbar at the bottom of the column area is included to allow the number of columns to extend to the right.

5.4.3 Column Design

The previous section looked at the Column area as a whole. The actual Column and its contents also need to be looked at.

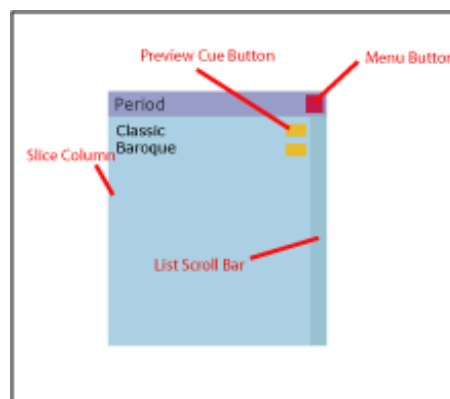


FIGURE 7: A Design for an Individual Column

When taking generalisation into account, the information within a column may not just be a simple list. It could be an image such as a map, which has different sections highlighted depending on the current Uri selection (in the same way as a list item selection). For this reason the column and its contents need to be separated.

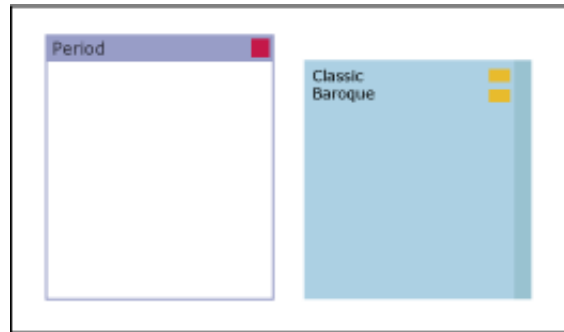


FIGURE 8: Column with Header and Menu Button

Figure 8 shows a column containing a header bar and a menu button, which are independent of the columns contents. In our Classical music case the content is a list. The menu button is used to provide all the functions that a user should be able to perform on a column. The List shown in Figure 7 is a custom list box where each item contains a label and a button to show the associated preview.

5.4.4 Information Box Design

The information area or information box is the section that displays data relevant to the users last known selection. This box in most cases needs to just display some text with some simple HTML formatting. It should look like the rest of the browser with a Column style header with a title of Information.



FIGURE 9: Initial Design of the Information Box

Figure 9 is an initial design of the information box, showing some dummy information about Ludwig van Beethoven. The information area needs to be scrollable, so that it does not get bigger when there is more information contained in it. The user will be able to scroll the information when they are interested in it.

5.4.5 Favourite List Design

The favourites list is the same as a normal column that contains a list, except that it does not have a menu button and is not part of the column layout. It needs to have a

header and a content area. There might be a case when generalising the UI where the Favourites List might not in fact be a list, so it should work on the same principle as the Column object.

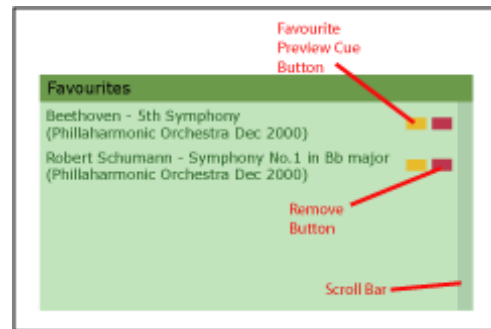


FIGURE 10: Favourites List Design

Figure 10 shows the Favourites List, where each item in the list is the same as a normal list with the addition of a remove button.

The initial design of the Favourites list was to have a normal list item that could be removed by double clicking the item. It was decided that this functionality was not user friendly, so a remove button was added.

5.4.6 Preview Cue Design

The idea of a Preview Cue is to popup a preview of the current item. In the classical music case this could be an example of the current composers work i.e. a piece of music. In the case of the movies domain, it could be an example of the current directors work, i.e. a small clip or trailer of a movie.

The Preview Cue popup should be able to contain any sort of media, or information to deal with any sort of domain. It should also be able to provide more than one preview item, so that a user can see a few examples of a particular composers work etc.

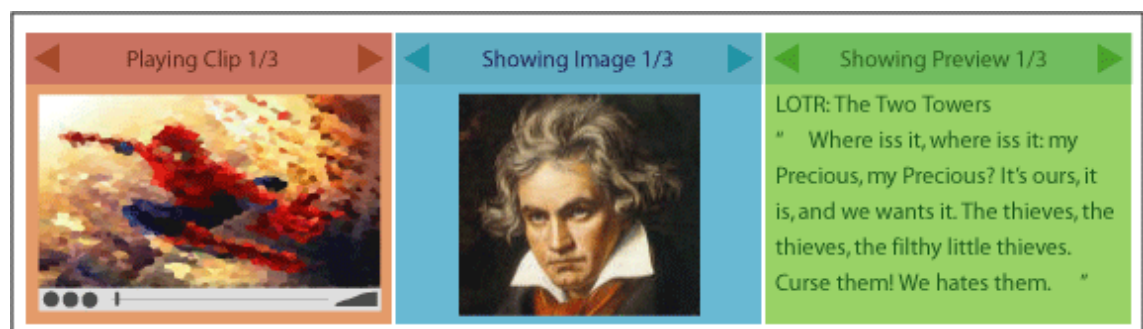


FIGURE 11: Preview Cue Designs

Figure 11 shows 3 Preview Cue designs, each with different types of content. There are navigation arrows at the top left and top right of the Preview Cue object, which

allow the user to navigate through each of the previews available. The text at the top indicates which of the previews the user is currently viewing.

5.4.7 Column Menu Design

Popup menus are a simple way to provide all the functions for the UI while keeping the interface simple and clean. The main mSpace functions will be performed on the Columns within the Column layout. Each individual column has a menu button associated with it that should popup a menu when the user clicks it.

The functions that the menu will need to provide are

- Close the current column
- Add a new column
 - To the Right of this one
 - To the Left of this one
- Move the current column
 - To the Left
 - To the Right

The function Add a new Column to the current layout should have a sub menu with a list of Columns that can be added into the layout.

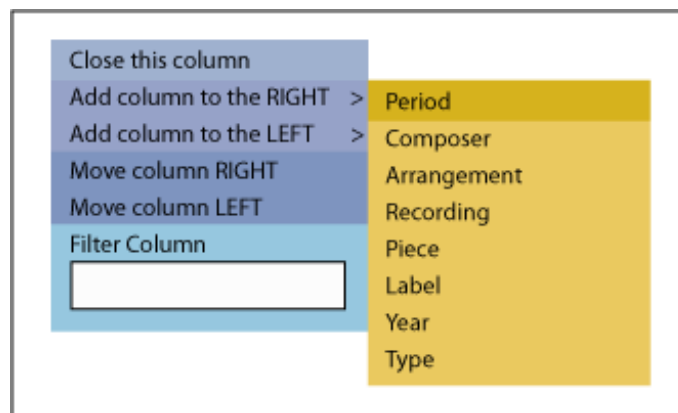


FIGURE 12: Popup Menu Design

Figure 12 shows a sample layout of the popup menu. It includes menu items and sub menus to provide the required functions. It also includes an extra feature (Filter Column) that is described in Future Work section.

5.4.8 Status Bar

The status bar was not included within the general layout design, but needs to be placed somewhere visible on the screen. It is a small area that can be used to display a message to the user. This message can flash to draw the users attention if it is important.



FIGURE 13: Status Bar Design

Figure 13 shows a sample of the Status Bar which will be placed between the Column layout and the Information Box/Favourites List.

5.5 Generalisation

The scope of this project can be massively enhanced by providing it with the ability to visualise data other than the initial set of classical music we have available to us. This process is known as generalising the system. Generalisation of data source changes the project from a feature-rich but limited classical music browser into a generic RDF browser; an application toolkit for the semantic web. This gives a real opportunity to provide the semantic web community with a tool that would be useful and (currently) unparalleled.

There are several advantages to implementing generalisation:

- Allows the visualisation of generic data.
- Amount of generalisation can be controlled. One could just generalise how the data is accessed, or go for the UI as well so that the way the data is visualised can also be changed.

There are costs associated with generalisation, although in general they do not overwhelm its utility:

- Extra code - although this can be limited by careful coding of the classical-music specific version.
- Adds time to start-up due to additional querying to find out how the data is formed.
- Prevents the addition of data-specific features, although people using the system could alter the code for themselves if they so desired.

5.5.1 Ontological Model For mSpace

The configuration of an mSpace installation is done through the definition of what we have called the “mSpaceModel”, which defines how the data is arranged, which dimensions exist, and how they are allowed to be ordered.

When designing this model, we have taken care to follow recommendations about the formalism[30]. Our model ontology defines classes for the Model, Columns and linked-list type classes for sorting, default layout and overall graph layout description, see Figure 14.

The Model class specifies the name of the model, the columns of the model, goal column and default column layout.

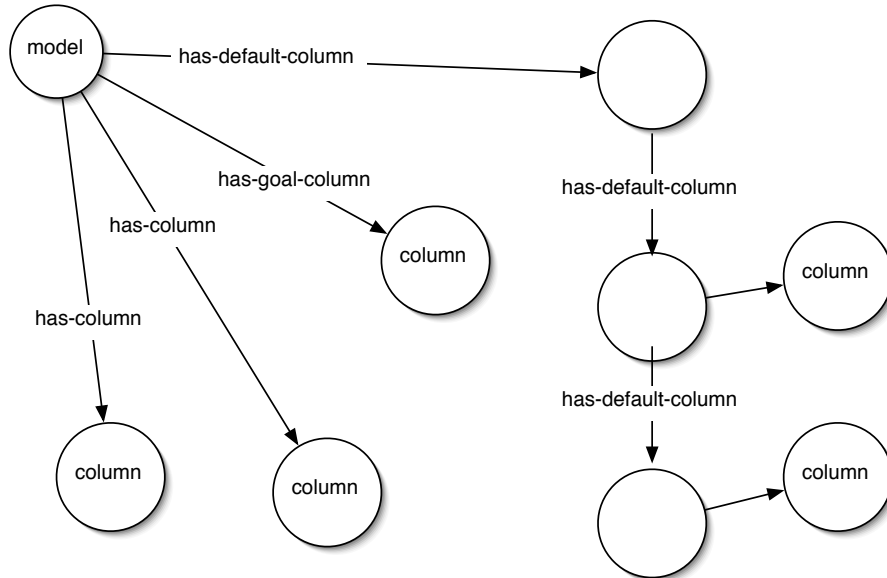


FIGURE 14: Graph Layout of the mSpaceModel

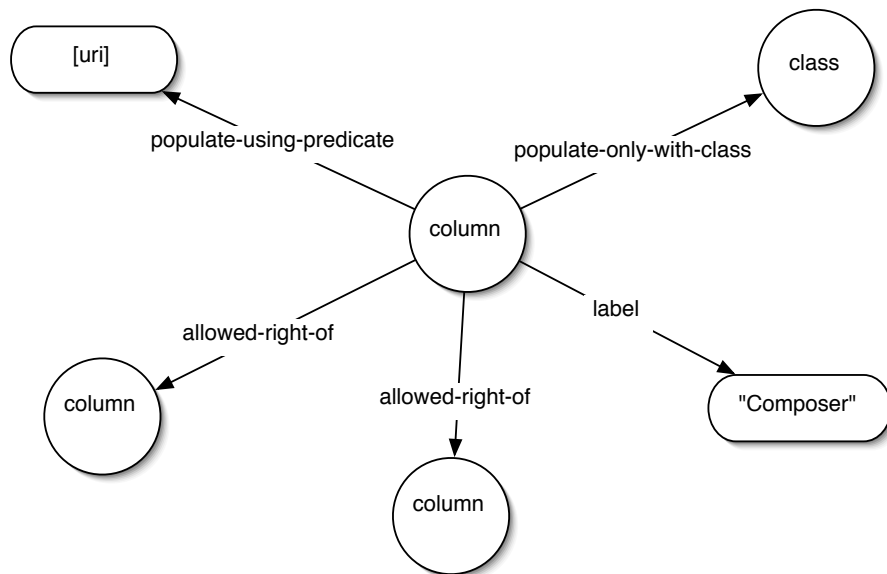


FIGURE 15: Column Class Layout

The Column class (see Figure 15) specifies the name of the column, the class with which to type-check the column items and a predicate list that details how the query should be formed through the use of a linked nodeset. Column objects also must specify which columns this column is allowed right of. This allows for a finite number of slices to be defined by the mSpace administrators. Columns may optionally provide a number of sort objects with which the column contents may be sorted, i.e. to sort Classical Music Eras by their dates.

5.5.2 UI Generalisation Theory

In many projects that involve a User Interface, a modular component type approach is often preferred. Components are usually created so they can be reused, and fitted together like a jigsaw.

Generalisation of mSpace as a project looks at allowing for different data sources or domains of information to be handled by the browser. Generalisation of the User Interface takes part of its requirement from this, but is weighted more towards allowing future development of components that can extend a users browsing ability.

The idea of generalising the mSpace UI is to provide a set of standard components that are capable of browsing the current domain, Classical music, but at the same time providing a platform that can be used in the future to develop features that will enhance aspects of different domains.

An example of where extensions may be required is the csAKTiveSpace project. This project contains an early version of an mSpace browser showing information about researchers and institutions throughout the UK. One feature is a map that allows the user to highlight sections of the UK in an effort to narrow down institutions or researchers bases on location. Development of a similar component for our mSpace UI would allow a user to navigate the CS AktiveSpace knowledge base in the way that was intended.

For the goal of the mSpace project, creating a platform with simple List components is as far as generalisation needs to go. The Classical Music domain does not require anything more than simple Lists of information, but this does not mean that extensions to the component library would not be usable. Having a Column that provides a timeline to the user that can be navigated or filtered may give the user a better understanding of when particular eras of Classical music came into effect.

5.5.3 UI Generalisation Implementation

The foundation for generalising the mSpace User Interface is using the HTML Document Object Model to create elements and manipulate them client side. As detailed in the UI Implementation section, the UI has been broken down into re-usable components that can all easily communicate with each other and fit together to provide the whole browsing experience.

Parent objects are a core means of communication between components, passing messages event handling and more. The method

```
getElementObject();
```


is used to access the DOM element that should be used to display the component. Each parent object does not need to know what the child object it is displaying is, only that certain methods are provided and that it will fit in correctly with the parents DOM element. Each ListItem, or component that extends it, must provide a root TR (table row) element to fit properly into the Lists DOM.

Components in the mSpace UI are plugged together in a hierarchy. This is how the HTML DOM works, where child elements are appened to the root element “document.body”. The root element in the mSpace UI is the page, or the mSpace controller object. All other components are used as children of this root element.

The mSpace UI can still be generalised further, as discussed in Section 10.3.

5.5.4 IMDB

In order to test the generality of the model and the suitability to all data domains, it was decided to apply the mSpace model to more than one domain. Given the possibility for effective use of preview cue technology through movie trailers and clips and the availability of the data from IMDB, the movie domain was chosen for this purpose (see Section 4.2).

RDF to define the mSpaceModel dimensions of the IMDB data was created and asserted into a new knowledge base with the IMDB data itself. A new configuration of mSpace was created that pointed to this new knowledge base (through the use of a new RDQL-handler URL).

5.5.5 Relevance of Previews

Each deployment of an mSpace interface will involve varying columns, each column could contains thousands of items, and each of those items needs a list of preview cues. Typically each preview will bear direct relevance to a goal item, though that is not essentially the case. With the classical music knowledgebase being used to test the solution developed in this project, all previews are pieces of music. It is however desirable to also allow other media types to be used as previews; photos or portraits of composers could be displayed as a preview for a solitary composer selection. The generalised nature of this mSpace solution means that the types of preview cues used can very simply be changed. All goal pieces in the classical music test slice have a preview which is the piece itself, in the real world there are many recordings of each of these pieces, however we don't have the entire knowledge of classical music in our example data set.

Preview cues are dependant on the column the preview is being requested for and all columns to the left of that column; in many cases there will be far too many preview cue

options to offer them all; this is especially the case when previews are required on the leftmost column. Each preview cue is an instance of a `PreviewableItem`; `PreviewableItems` have filenames which might resolve to the image, audio track or movie clip. Each `PreviewableItem` will bear some relevance to each thing which it could potentially be used as a preview for; the relevance is specified as the percentage the `PreviewableItem` is relevant to the resource and the identifier for the resource. [diagram]. The relevance information should be added to the knowledge base when each track is added, and then a script can be run daily which chooses the most relevant pieces. These will then become the current previews for a particular resource. If the preview cue selections were based entirely on the relevance information then they wouldn't change unless the relevance data was manually updated, therefore the relevance data is augmented with user ratings and play counts to better determine which `PreviewableItems` should currently be used for each resource.

6 User Interface Implementation

6.1 Implementation

To make the UI more generalised and the code easier to understand, JavaScripts Object Oriented capabilities were utilised. Each area of the UI is wrapped up within an Object, and all these objects communicate with each other.

Using the HTML DOM (Document Object Model) provides us with the ability to tie in the custom Objects with the pages document structure and HTML elements. There are many functions that the DOM provides for manipulating HTML and elements, as well as providing event handling abilities.

The following sections outline the structure of the objects and how they communicate, as well as investigating some of the specifics of each object.

6.1.1 Simple Object Class Diagram

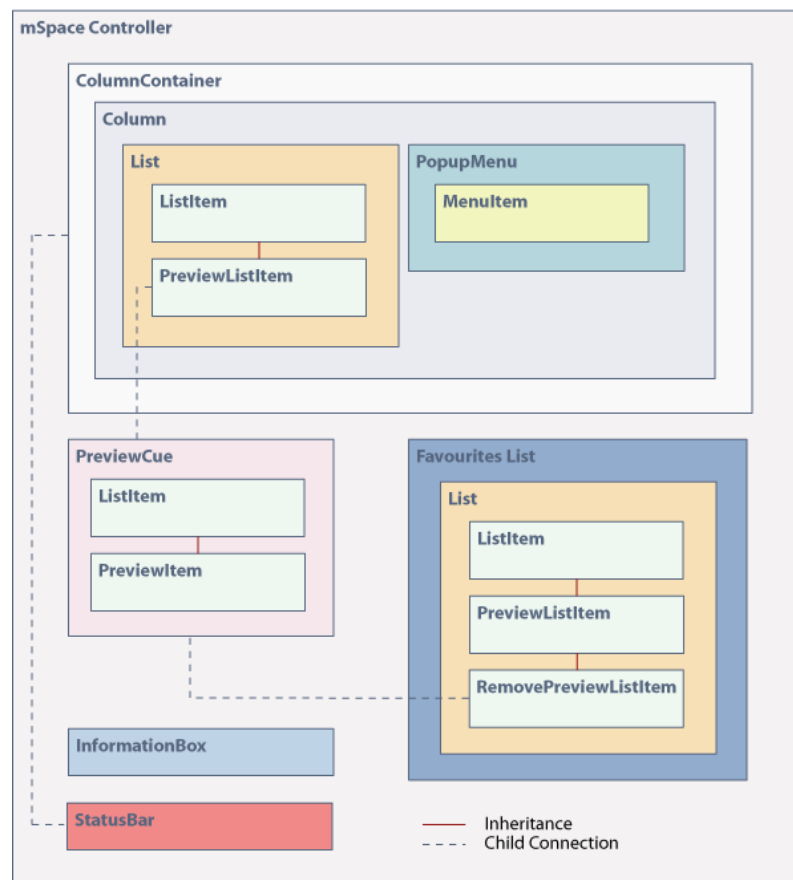


FIGURE 16: Class Organisation Diagram

Figure 16 is a simple class diagram that shows how objects are related and interact with each other. It does not include any method or variable declarations.

6.1.2 JavaScript Objects

The mSpace User Interface has been broken down into the following objects, each defined in its own .js file. Each of the Objects in the class diagram above is represented by a .js file, and there is a generic.js script which contains functions that are used throughout the UI.

Each file contains the constructor for the object. Within this constructor all variables are defined and initialised, and all methods for the object are assigned to functions within the file. Each method is prefixed with the name of the object and an underscore, to make it unique within the project. For example, the onClick method for the Column object, would be defined within Column.js as:

Function *Column_onClick()*

6.1.3 HTML Document Object Model

The HTML DOM is the foundation of the mSpace UI. It allows for html page elements to be stored as variables, accessed and rearranged within our JavaScript code.

Creating a node that can be stored as a variable is as simple as calling

```
document.createElement(elementType);
```

This will return a variable that can be manipulated using the DOM.

Rearranging html elements is easy using the following two methods:

```
element.appendChild(childElement); element.removeChild(childElement);
```

Each of our objects is built on the idea that it has a root HTML element that is used by its parent for appending to the pages DOM, and essentially displaying on the screen.

Figure 17 shows a basic DOM layout, where the root TABLE element is stored within our JavaScript object, and can be used by its parent to display. The parent of this object will call the function

```
getElementObject();
```

which will return a reference to the TABLE element in Figure 17. This can then be used in the

```
element.appendChild(TableElement);
```

call, which will add the table into the pages DOM.

Similarly there is a link in Figure 17 to the Childs Display Node. If the object represented in the diagram was to call

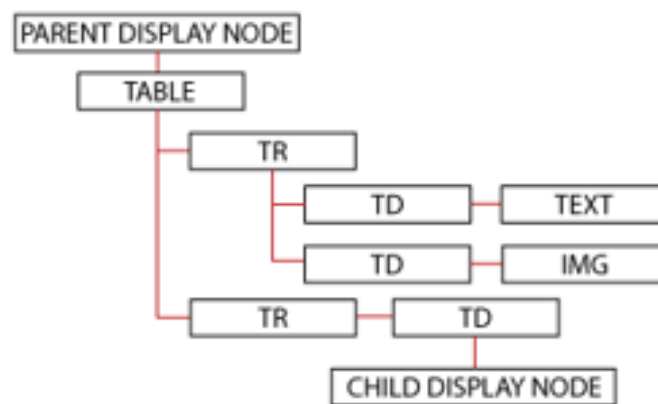


FIGURE 17: Basic DOM Layout

`element.appendChild(Element);`

on the table cell element (TD) and pass it the child's display node, then this would add the child element into the page's DOM, which is essentially displaying the object on the page if the main root element is the page's document itself.

6.1.4 Column Object

The Column object is one of the key components of the mSpace UI. It provides a column header describing the data, a content area for drawing data to and also a menu button. The Menu button is used for the popup menu that contains the functions to rearrange columns.

Like many of the objects in the UI, the constructor for the Column object accepts a required parent parameter, which specifies an object that will receive messages from the column, and essentially contain the column. In our mSpace browsers case, the parent object is a ColumnContainer object.

There are several optional parameters that can be supplied to the Column object which affect the way it appears. These parameters *width*, *widthUnit*, *height*, *heightUnit*, *colHeaderStyle*, *colHeaderBtnStyle* and *colContentStyle* can be left out and will be set to the default parameters. They are all fairly self explanatory; the style parameters are used to change the CSS style of various parts of the Column object, and the width and height parameters are used to set the width and height respectively.

The width and height can also be set using the Columns *setWidth* and *setHeight* methods. Both methods accept an integer value and have an optional unit value, like the constructor that specifies the type of value that is to be used. Possible values include Pixel Units (px) and Percentage (%).

The constructor creates an internal table object, then creates a header row and inserts the title of the column and a menu button in this row. The title of the column is passed on construction as *description*.

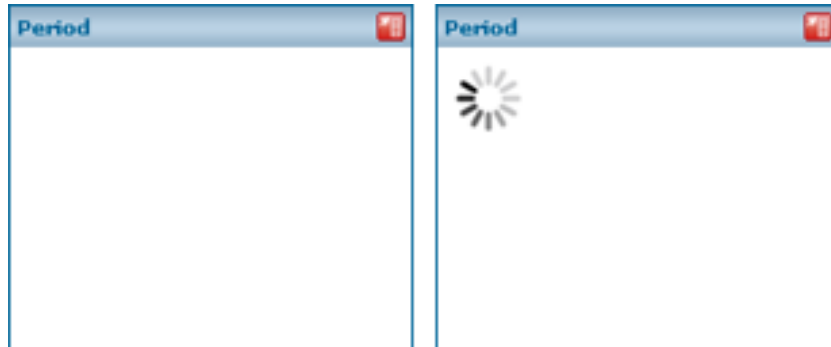


FIGURE 18: Two Column Objects

Figure 18 is a screenshot of two column objects. One has no internal object to display, and another has a loading animation. The loading animation has been set by calling the *setLoadingStatus(true)* method on the column object. The columns are using the default style provided by the column object, which is defined in *mspace.css*.

One other parameter which is required is the *columnObject* parameter, which represents the child element that is to be displayed within the column.

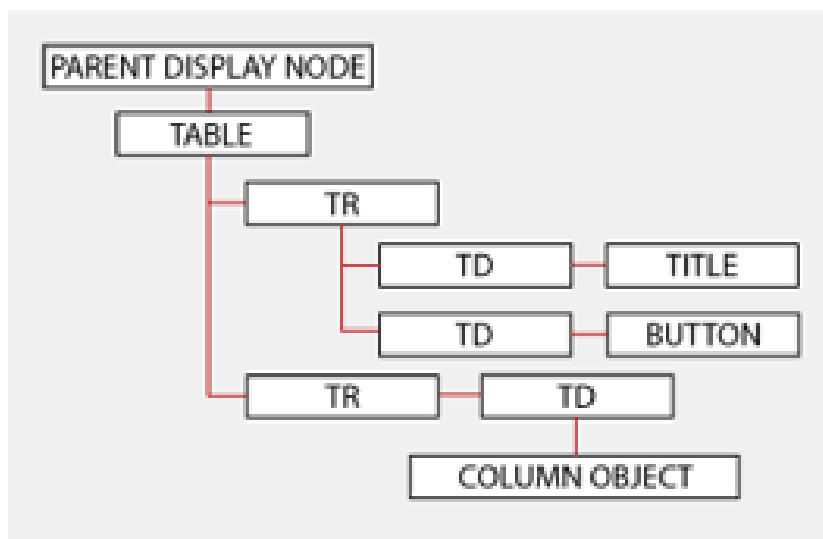


FIGURE 19: Column DOM

Figure 19 shows the Document Object Model Structure of the Column Object. The parent display node in this case will be a display node of the ColumnContainer.

6.1.5 Column Container Object

The ColumnContainer object handles all the columns within the current slice. It is responsible for displaying the columns, and rearranging them if necessary. Its constructor takes a parent parameter which it uses to draw itself to. The parent object should provide a getElementObject method which will return the HTML element that the ColumnContainer can append itself to. The parent also receives calls from the ColumnContainer such as onChange or mouse events. The optional size and style parameters are used in the same way as for the Column Object.

The ColumnContainer object has 3 arrays which are used for storing the Columns used within the mSpace domain.

These are:

allColumns Contains all the columns

displayedColumns Contains only the columns actually visible

nonDisplayedColumns All columns not actually visible

`allColumns = displayedColumns U nonDisplayedColumns`

To add a new Column to the ColumnContainer, the method

`addColumn(column, display);`

is used. The column parameter is the actual column to add to the container, and the display parameter determines whether the Column is put into the *displayedColumns* array and displayed straight away.

The ColumnContainer object provides methods that also handle the rearranging of columns. When a menu item is clicked within a Column, it will call the appropriate method within the ColumnContainer.

- `moveColumnLeft(column)`
- `moveColumnRight(column)`
- `addColumnLeft(newColumn, column)`
- `addColumnRight(newColumn, column)`
- `closeColumn(column)`

The *column* parameter is the Column object that the function call originated from, and the *newColumn* parameter used in the add functions is the new Column to actually add to the layout. When adding a new Column, the Column that the call originated from is the base column.

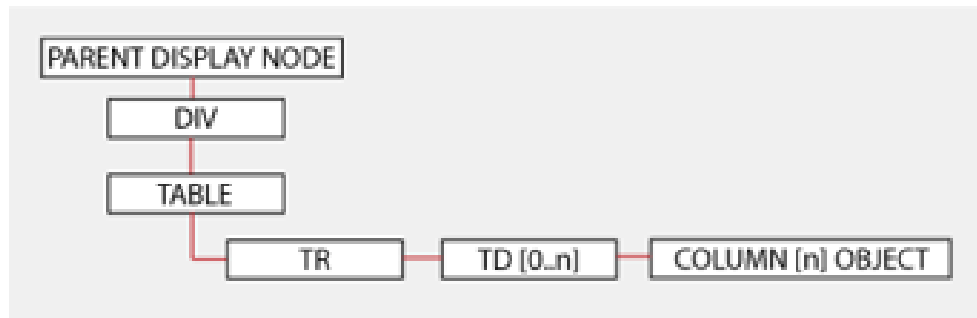


FIGURE 20: ColumnContainer DOM

Figure 20 shows the DOM layout of the Column Container. The root element is a DIV element, which allows scrolling when its child table gets to large. The table element has one row, which contains a TD cell for each of the columns that is to be displayed.

Whenever the column layout changes, the repaint function is called, which calculates the width of each column based on the width of the ColumnContainer. It sets the width of each column, but if the width is less than the constant *minWidth* defined in the Column object, then the TABLE will be larger than its parent DIV, forcing the DIV to scroll.

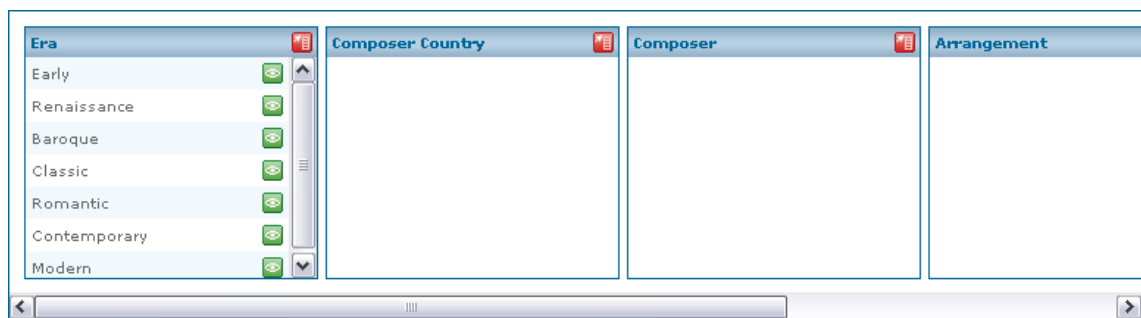


FIGURE 21: ColumnContainer with Columns Contained

Figure 21 is a screenshot of the ColumnContainer with a number of Columns contained within it. It shows that the width of all the Columns is wider than the ColumnContainers DIV element, so the DIV element has added a horizontal scroll bar.

6.1.6 Favourites List Object

The Favourite List object is much like the standard Column object, the main difference being that the child display object is a List object that cannot be changed. It also does not provide a menu button, since it is not part of the actual Column layout.

The constructor is fairly similar to that of the Column object. It takes a parent parameter, which is used to send event messages, such as mouse events or item removed events. It also takes a surface parameter which is used for appending the root HTML element to, and optional size and style parameters used in the same way as in the Column object.

The parent object is required to provide a method which returns a `RemovePreviewListItem` object based on a URI.

```
parent.getFavourite(uri);
```

This method is called when an item needs to be added to the Favourites List, through user interaction or when the page loads and the Favourites List state is loaded.

The Favourites List is responsible for its own statefulness. It saves the URI of each element that is contained within the child List object, and writes these values to a cookie. The cookie is then loaded again when the page is displayed and parsed to read all the saved URIs. The *getFavourite* method is called for each URI that has been saved, and the returned object is added to the child List.

The *addItem* function is called by the Favourite Lists parent, when the user requests an item to be added to the List. It contains a URI, which is used in the call to *getFavourite* to display the object in the List.

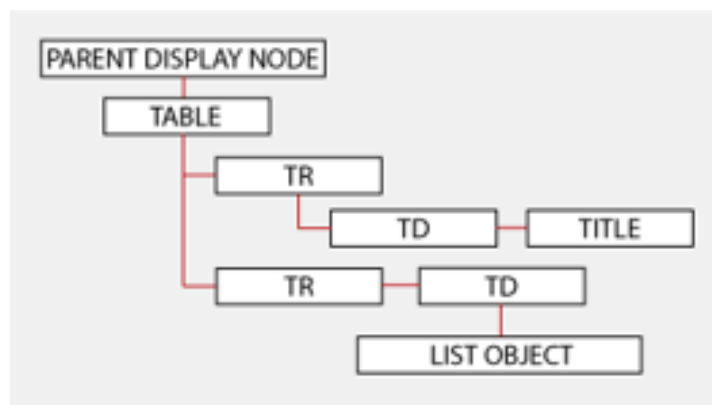


FIGURE 22: Favourite List DOM

Figure 22 shows the DOM layout for the Favourite List. It is almost identical to the Columns DOM layout, except that it does not provide menu button.







Favourites	
Concerto For Violoncello And Strings In D Major	 
Fasch. Concerto For Tumpet Oboes & Strings In D Major II. Largo	 
Allegro vivace - Trio. Un poco pié tranquillo - Vivace. A tempo	 

FIGURE 23: Populated Favourites List

Figure 23 is a screenshot of the Favourites List, containing 3 items that have been added to the child List object.

6.1.7 Information Box

The information box is a simple area to display information in text form. The `InformationBox` is the only object that does not conform to the DOM standard of the other objects. This is because it was an item coded very early on in the project that has not been altered since. It is part of the future work to re-write this object to be similar to the other objects within the UI.

It uses the same layout as the Favourite List and the Column object, with a Column header containing a title, and then a content area used for loading the information. There are two methods provided that load information into the content area:

```
displayText(text);
```

This method simply displays the string within the information box, using the inner-HTML method of the content areas html element.

```
display(url);
```

The `display` method takes a URL, and loads the page into the content area of the information box.

The `display` method uses two different techniques for displaying the destination page, depending on the browser. See the code in Appendix A for details on the methods used.

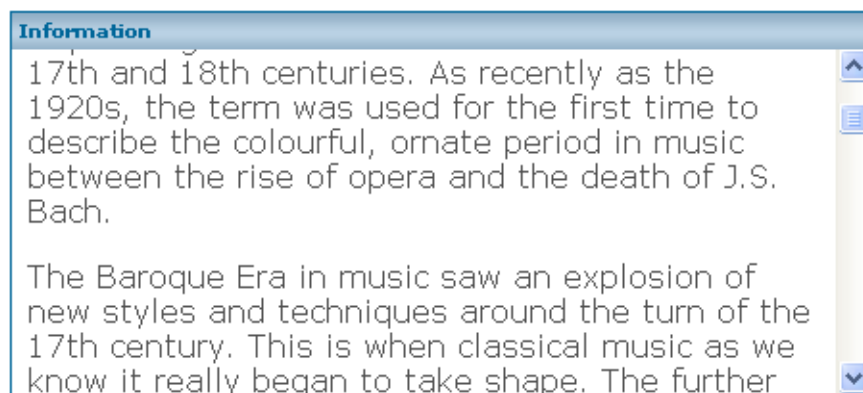


FIGURE 24: Information Box

Figure 24 is a screenshot of the information box, displaying some information about a particular era of Classical music.

6.1.8 List Object

The List object is one of the key components of the mSpace UI. It is the object that displays the information in our Classical music implementation. The Column object can be used to display any object that displays data, but in this project the List is the object of choice.

The List object is actually fairly simple, particularly the DOM layout. It leaves the drawing to the objects that represent List Items. The main purpose of the List object is to provide a scrollable area to draw on, and an object that handles the list items, including any events that occur.

As normal it is passed a parent parameter on construction. This will usually be the Column object, but in the case of the Favourite List it is not. This parameter is somewhat redundant when used with the Column object, because usually the List is created before the Column and passed into the Column object on construction. The parent is therefore re-set within the Columns Constructor. The other required parameter is the URI of the List. This is part of the mSpace model, and is used to specify what data the List actually represents.

The other optional parameters are for the size of the List object, which is all taken care of at a later stage by the parent Column object when the Column is resized by the Column Container

To add an item to the list, the method

```
list.addItem(ListItem);
```

is called. This takes an object that extends the ListItem object, which provides all the functionality needed for a simple list item. There are 3 other types of list item that use the ListItem object as a base. These are discussed in later sections.

All the ListItem objects are added to an *items* array. The ListItem object handles appending its display element to the Lists TABLE element. It uses a function within the List called

```
getDisplayObject();
```

which returns the Lists TABLE element. The List object also has two methods for setting and determining which item within the List is selected. These methods are

```
listobject.setUri(URI); URI =listobject.getUri();
```

The URI parameter is the RDF identifier for the particular item in the 3store. There is also a method to check to see if the List has a particular item.

```
containsUri(URI);
```

returns true if the item with the specified URI is in the List, false if it is not.

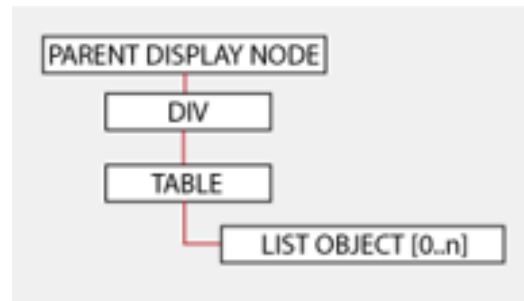


FIGURE 25: List DOM

Figure 25 shows the DOM layout of the List object. It contains just a scrollable DIV, with a TABLE child element. Each child List Item creates its own TR element which is appended to the Lists TABLE element.

The DIV element uses the size parameters of the List. If the number of elements in the List increases the size of the child TABLE, the DIV will display a vertical scrollbar.



FIGURE 26: List Object as child of Era Column

Figure 26 is a screenshot of a List object set as the child of the Era Column. The list items are PreviewListItem objects, and since there are too many to fit onto the visible surface of the List the vertical scrollbar has appeared.

6.1.9 List Item Object

The list item object is part of the List object. It is responsible for appending itself to the display area of the List, as well as handling any events that occur and passing them onto its parent List. The basic ListItem object is just a TABLE row containing a piece of text. It has an associated URI, and also different styles to change the appearance of the item.

A ListItem can also have a popularity rating applied to it, which is used to highlight the ListItem in another colour so that it is more visible to the user. This popularity rating

is between 0 and 1, where 1 would colour the ListItem completely, but anything above 0.5 would only tint the ListItem. Values below 0.5 are not highlighted.

The ListItem constructor takes 3 required parameters. As usual there is a parent parameter, which in most cases is the List object that the item uses to append it to the DOM, and also pass on events.

The URI parameter is the RDF identifier associated with the data that the ListItem represents. This should be unique to each item within a list. The description parameter is the text that will be displayed, and is usually a descriptive word or sentence about the data the item represents.

There are several optional style parameters that can be passed to the constructor. These are applied the ListItem when it is the 4 following states:

- List Item is an odd item in the List
- List Item is an even item in the List
- List Item is selected
- List Item has a popularity greater than 0.5

There is also a parameter that can specify the popularity colour to use. This parameter should be a simple array with 3 integer values between 0-255 for each component of Red, Green and Blue. This array is used to calculate the ListItems colour when the popularity rating is between 0.5 and 1.

The ListItem handles its own mouse events. When the user clicks on the item it informs its parent object, usually a List, and sets its style to the selected style. The ListItem also passes on mouse over and out events, and allows a 300ms delay on any click to see if the user is double clicking. The List object is responsible for making sure that there is only one selected item within the list.

The ListItem can also be set to be an odd or even item. When there are many items within a list, it is sometimes easier to distinguish each item if the colours alternate. This is where the odd or even parameters come into play. The ListItem will choose an odd or even style for itself when idle (not selected) depending on this parameter. This allows the parent object to alternate the colours for each of its list items.

Figure 27 is the DOM layout for the ListItem. It is a simple TABLE row, with one cell and some TEXT. The ListItem assumes that its parent display node is a TABLE element.

Figure 28 is an example of a Column and List combination where only the simple ListItem is used. This shows the odd and even style for each item in use, and also a ListItem being selected.

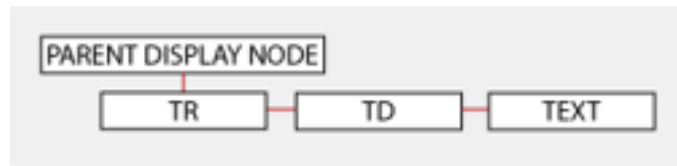


FIGURE 27: List Item DOM



FIGURE 28: Simple List Item

6.1.10 Menu Item Object

The MenuItem object is very like the ListItem. It is responsible for appending itself to its parents display element and handling events. In most cases the parent is a PopupMenu. The MenuItem object has an action associated with it, which is a unique identifier much like the ListItems URI. It can also have a PopupMenu item associated with it, so it acts like a sub menu.

The constructor of the MenuItem has 3 required parameters. The parent parameter specifies the object to append to and also receive any events. The action parameter is a unique string that identifies the purpose of the item, and the text parameter is the string to display that informs that user what the item does.

There are 3 optional style parameters which cater for an idle MenuItem, a MenuItem when the mouse is positioned over it, and for when the MenuItem is just a separator. If the text parameter passed in on construction is a single hyphen (-), then the MenuItem will act as a separator, in that it does not handle any events and uses the separator style. This allows users to create divides within a PopupMenu.

The method

```
addSubMenu(menu);
```

is used to add a PopupMenu to the MenuItem. This menu will then be opened when the mouse moves over the current MenuItem.

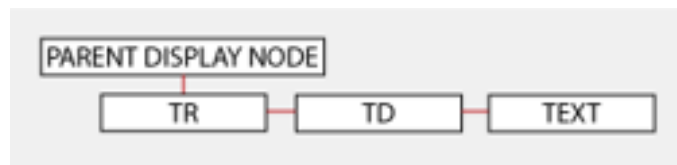


FIGURE 29: Menu Item DOM

Figure 29 is the DOM layout for the MenuItem object. It is identical to the DOM layout for a ListItem object.

Close This Column	
Add Column To The LEFT	
Add Column To The RIGHT	Arrangement
Move This Column To The LEFT	Composer Death Date
Move This Column To The RIGHT	Composer Country

FIGURE 30: Popup Menu with Several MenuItems

Figure 30 shows a screenshot of a PopupMenu using several MenuItems. Two of these are separators and one has an associated sub menu, which is displayed. The MenuItem with the associated sub menu is also highlighted due to the mouse rolling onto the item.

6.1.11 Popup Menu Object

The Popup Menu object provides a container for all the MenuItems that are added to it. It handles messages such as mouse clicks that are passed from the MenuItems, and provides methods to open the Menu at particular co-ordinates or close it. The constructor is very simple. It takes a parent argument to pass on any mouse events or item clicked events, and an optional style parameter which defines the css style of the menus DIV element.

Once a PopupMenu object has been created you can add an item to it by calling the

```
popupMenu.addItem(MenuItem);
```

method, which takes a MenuItem object and will add it to the internal array of items. The MenuItem object provides the standard method for getting its root HTML element, which the PopupMenu uses to display each item. To open the PopupMenu the parent object (or any other object) can call the method

```
popupMenu.openMenu(xPos, yPos);
```

This method takes the screen co-ordinates that the menu should appear at. These co-ordinates set the Top-Left position of the menu. All sub menus open up to the right of their parent MenuItem. Similarly you call the

```
popupMenu.closeMenu();
```

to close a PopupMenu that is open.

The PopupMenu object handles its own display status. It receives mouse over and out messages from all its MenuItem objects, which allow it to hide itself if the user is no longer using it. All mouse clicks that are received by the PopupMenu are passed onto its parent object.

If a PopupMenu's parent object is a MenuItem it is classified as a sub menu. Sub Menus are just normal popup menus that are controlled by a MenuItem rather than a button. Any messages from the MenuItems within a sub menu are passed along the chain, and eventually treated as if they came from the root PopupMenu.

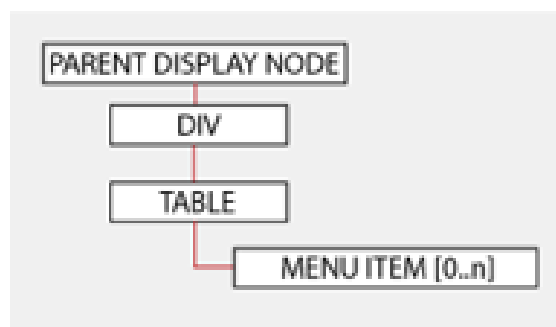


FIGURE 31: Popup Menu DOM

Figure 31 shows the DOM layout of the PopupMenu object. It is identical to the List object except that the root DIV element does not allow for scrolling.

6.1.12 Preview Cue Object

The PreviewCue object is used to display sample previews for particular goal data. In the classical music case this is short sound previews for particular pieces. The PreviewCue object needs to have an area to display information, such as an embedded media controller, and also an area that the user can use to scroll between the different previews available, if there is more than one.

The initial design of the PreviewCue provided two buttons to navigate through each of the previews. This design has been changed since implementation to a list of items that the user can click on as it was felt this method was more intuitive.

The constructor of the PreviewCue object requires a parent object, which like all other parent objects is used to receive events and other messages. This parent object also needs to provide a method to actually receive the preview cue items. In our Classical music browser the Controller is responsible for providing this method; the PreviewCues parent is therefore not the ListItem that actually opens it up.

The PreviewCue object also uses the root document.body element to append its own root html element. This means that all preview cues are automatically added to the pages DOM when they are created.

The PreviewCue can also be pre-cached with an array of preview items. This is not a feature used in the Classical music mSpace browser as it would require a lot of data to be loaded before the browser can be used. It is possible that this could be used in special circumstances.

As usual there is also an optional style parameter which can be used to add a custom style to the PreviewCues DIV element.

To open the PreviewCue object there is a display method which takes two screen coordinates, much like the PopupMenu object. This will open the PreviewCue object with the first preview item selected, unless the user has already opened the PreviewCue, in which case the last item selected will be in focus.

Closing the PreviewCue is done using the provided close method. The opening and closing of PreviewCue objects is handled by the PreviewListItems that use the PreviewCues. When a PreviewCue object is closed, it sets the content of its preview area to an empty string. This is to avoid problems with embedded media, which will continue to play even when the PreviewCue is not visible.

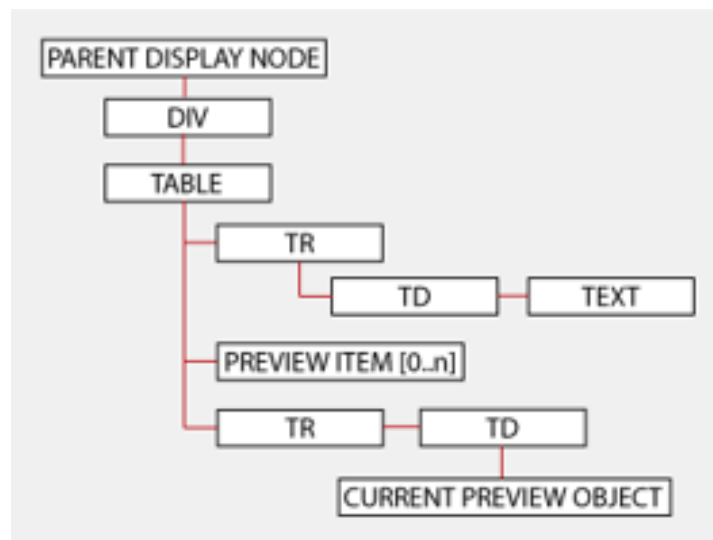


FIGURE 32: Preview Cue DOM

Figure 32 shows the PreviewCue objects DOM layout. The root DIV element is not scrollable; thus the whole preview cue is always visible. The DIV has a child TABLE which contains one row to display a title, and then a number of rows for each of the previews that are associated. There is one more row with one cell which is used for actually displaying the current previews text.

The PreviewCue implements a custom List, using the PreviewItem which is an extension of the ListItem object. The PreviewCue object acts as a parent and provides some of the same methods that the List object does, so as to handle the PreviewItems that are displayed. These PreviewItems each provide a row to the TABLE element.

When a user clicks on one of these PreviewItems the PreviewCue will replace the current preview object with the preview object of the item that was clicked. In the Classical music case the preview object is an EMBED element that embeds an mp3 file for the browser to play with its default plugin.

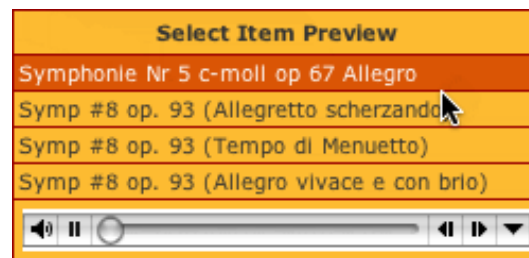


FIGURE 33: Preview Cue Object

Figure 33 is a screenshot of an open PreviewCue object which has 4 preview items associated with it. The one that is currently playing has an embedded QuickTime controller in the main preview area, which can be used to search through the playing song. The embedded content is not part of the UI, and should be controlled by the associated data in the 3store.

6.1.13 Preview Item Object

This object is an extension of the ListItem object with its own default styles and one additional internal variable content, used by the PreviewCue object for displaying in its preview content area. The only difference between this object and the ListItem object is the content parameter which is passed in. In our Classical music browser the content is an EMBED element which specifies the .mp3 for previewing. For more information about this item, refer to the Section 6.1.9 detailing the ListItem object.

6.1.14 Preview List Item Object

The Preview List Item object is an extension to the ListItem object. The most notable difference with this object is that it has a preview button on each item, which when hovered over with the mouse opens up a PreviewCue object. The constructor is almost identical to the ListItems constructor. There is, however, one extra parameter which is required. This parameter is a PreviewCue object which is an object that provides a method to get preview items based on the items URI.

When the user first moves the mouse over the preview button, the `PreviewListItem` looks to see if it has loaded the preview items, and if it has not it will call the method `getPreviewItems(ListItem)`;

which will return an array of `PreviewItems` which can be passed to a new `PreviewCue` object. This `PreviewCue` is then opened.

The `PreviewListItem` leaves most of the event handling to its base object the `ListItem`. It does deal with event handling for its preview button, which needs to open up the associated `PreviewCue` when hovered over or even clicked.

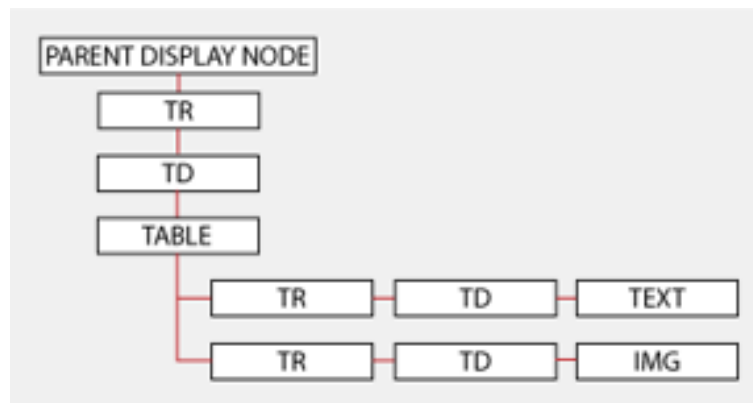


FIGURE 34: `PreviewListItem` DOM

Figure 34 is the `PreviewListItem` DOM layout. It extends the `ListItem` by adding an extra `TABLE` element, which has 2 cells. One contains the normal item label, and the other has an `IMAGE` element. The mouse over and click events for the image are handled by the `PreviewListItem` and are used to display the associated `PreviewCue`.



FIGURE 35: Column and List combination with `PreviewListItems`

Figure 35 shows a Column and List combination which uses `PreviewListItems`. The green button is used to open up the associated `PreviewCue`. Most of the styling is inherited from the `ListItem`.

6.1.15 Removable Preview List Item Object

The `RemovePreviewListItem` adds an extra button to the `PreviewListButton` for use in the Favourites List. This button is used to remove an item from the List which is required for a Favourites List Item. The Lists parent, a Favourite List object, receives an on click event with a type of *Constant_removeItemClick* (defined in `generic.js`), which it takes the appropriate action to remove the item.

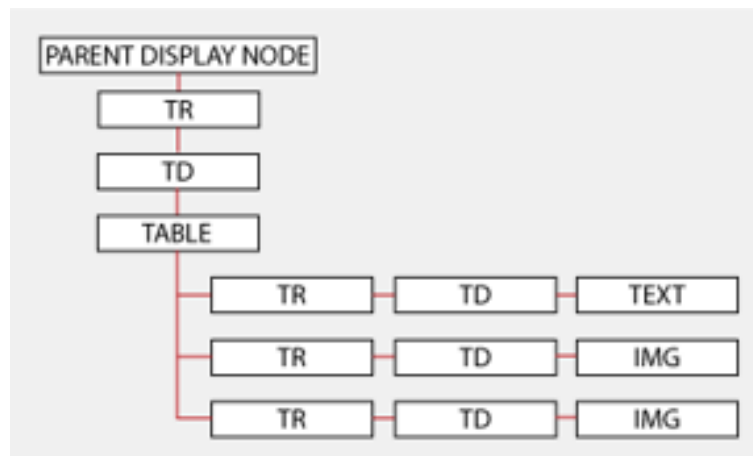


FIGURE 36: Removable Preview List Item DOM

Figure 36 is the DOM layout for the `RemovablePreviewListItem`. It adds an extra table cell containing an image for removing the item from the list.

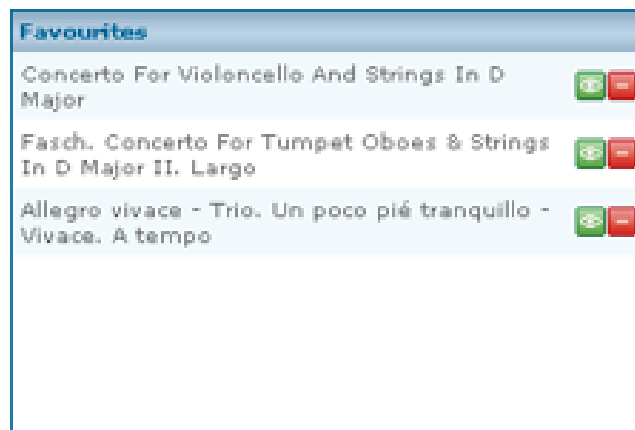


FIGURE 37: Favourites List

Figure 37 shows the `FavouritesList` using the `RemovePreviewListItem`.

6.1.16 Status Bar Object

The `StatusBar` object is used to display messages to the user. The constructor has two required parameters. The parent parameter is used in the same way as throughout the

mSpace UI. The parent receives an `onClick` message with an `action` parameter that gets set by the method that calls the method to set the message in the status bar. There is also a `surface` parameter which is an `html` element that the status bar can append itself to. This is usually an element within the pages DOM.

The method

```
setStatus(action, text, highlight);
```

is used to set the current message. The `action` parameter is used for `onClick` events when the user clicks anywhere on the status bar. The `text` parameter is the string to display in the status bar area and the `highlight` parameter is a Boolean to indicate if the message should flash for a few seconds (to get the users attention).

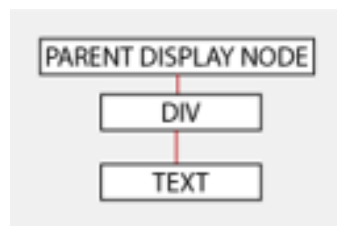


FIGURE 38: Status Bar DOM

Figure 38 is the DOM layout of the Status Bar object.



FIGURE 39: Status Bar DOM

Figure 39 shows two screens of the Status Bar object. One of the screens is the status bar in a normal mode, and the other shows the highlight mode.

6.1.17 Generic Functions & Variables

The file *generic.js* is a JavaScript file that contains some constants and functions that are used throughout the mSpace UI.

The following constants are used for events passed throughout the UI and to the mSpace Controller to identify the type of event. The name of some of the items is misleading, as they refer to list objects when in fact they are used for Column events. This is an item that needs consideration in the future.

```

Constant_listItemClick = "LIST_ITEM_CLICK";
Constant_listItemDoubleClick = "LIST_ITEM_DBL_CLICK";
Constant_listItemChanged = "LIST_ITEM_CHANGED";
Constant_listAdded = "LIST_ADDED";
  
```

```

Constant_listRemoved = "LIST_REMOVED";
Constant_listMoved = "LIST_MOVED";
Constant_listItemOver = "LIST_ITEM_OVER";
Constant_listItemOut = "LIST_ITEM_OUT";
Constant_closeBtnClick = "CLOSE_BTN_CLICK";
Constant_removeItemClick = "CLOSE_BTN_CLICK";

```

The following constants are used to identify each JavaScript object. All the UI objects have an objectType parameter which can be used to identify the type the object is.

```

Constant_columObject = "CONSTANT_COLUMN_OBJ";
Constant_listObject = "CONSTANT_LIST_OBJ";
Constant_listItemObject = "CONSTANT_LISTITEM_OBJ";
Constant_previewListItemObject = "CONSTANT_PREVIEWLISTITEM_OBJ";
Constant_preivewItemObject = "CONSTANT_PREVIEWWITEM_OBJ";
Constant_preivewCueObject = "CONSTANT_PREVIEWWCUE_OBJ";
Constant_menuObject = "CONSTANT_MENU_OBJ";
Constant_menuItemObject = "CONSTANT_MENUITEM_OBJ";
Constant_favListObject = "CONSTANT_FAVLIST_OBJ";
Constant_removePreviewListItemObject = "CONSTANT_REMOVE_PREV_LIST_ITEM";

```

The following are functions used throughout the mSpace UI.

findPosX(element) & *findPosY(element)* These two methods are used to find the actual screen position in pixels of any html DOM element. This is used for positioning some of the popup DIV elements.

srcDownload(url) Used by the Internet Explorer browser to load a URL into the Information Box.

associateObjWithEvent(obj, methodName) Used by all objects to register event handlers for html DOM elements.

setCookie(name, value, expires, path, domain, secure) Used to set a cookie on the client system.

getCookie(name) Used to retrieve a cookie on the client system.

alphaColour(r, g, b, a) Used to blend an RGB colour with white using the alpha component a.

Figure 42 shows how mouse click events are passed from MenuItem objects to the Column Object. There is a loop between MenuItem and PopupMenu which highlights the passing of events through sub menus.

6.2.1 User Interface Bugs & Features

There is one main feature of the mSpace UI that arose from a bug in the early stages of development.

There was bug on Mac OS X systems where scroll bars within a browser were always displayed above absolutely positioned DIV elements. This meant that if the ColumnContainer or List objects had to many elements, and painted scroll bars, all preview cue and menu popups were slightly obscured. This has been fixed by adding a hide scroll bars method in each object that would display scroll bars. A method call is passed through to the ColumnContainer by any object that wishes to hide the scroll bars, and the ColumnContainer then calls the method on any Columns that are displayed, and this in turn calls the child List object.

When the object that is displaying scrollbars using scrolling DIV elements receives a hide scroll bars call, it sets the overflow attribute of the DIV element to hidden which will remove any scroll bars. The same process is then used to show scroll bars when any popup objects have been closed.

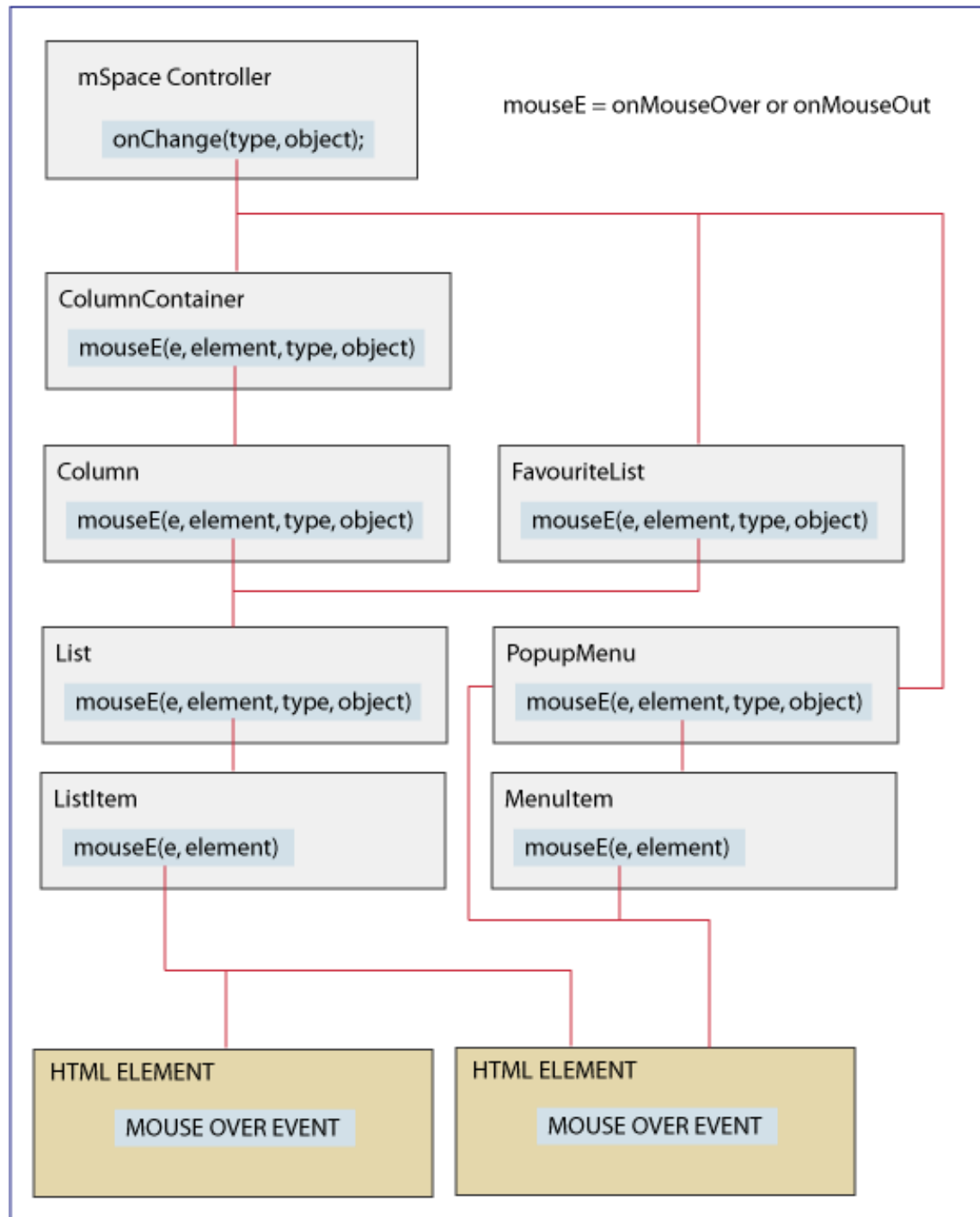


FIGURE 41: Mouse Movement Event Chain

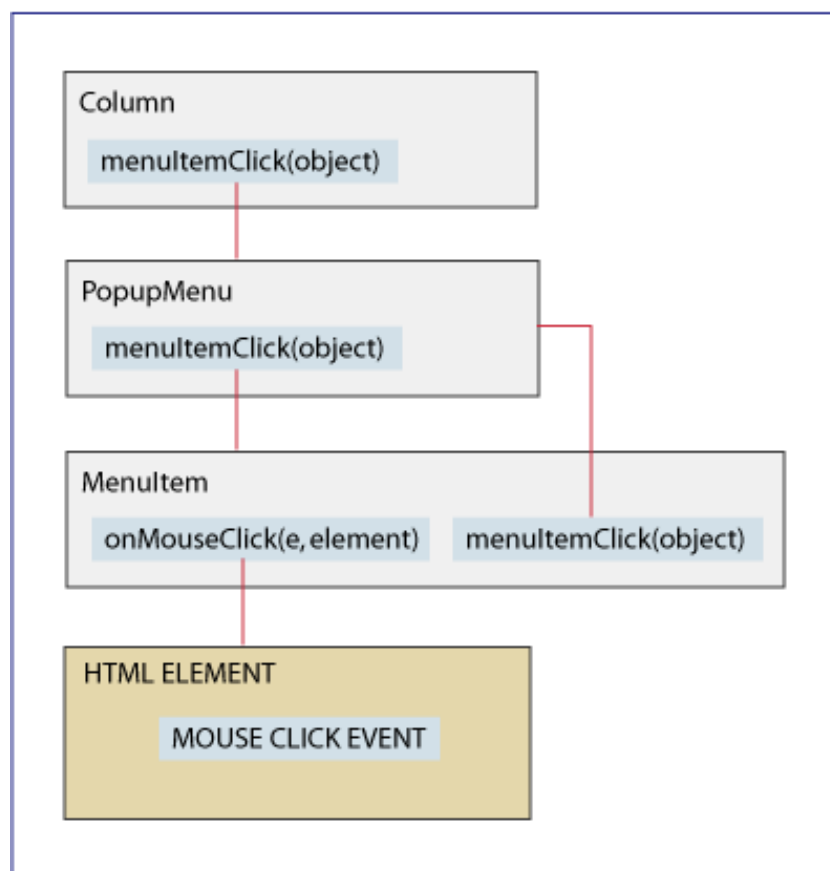


FIGURE 42: Menu Mouse Event Chain

7 Controller Implementation

This section describes the work involved in implementing the controller or mid section. This is client side code that populates the UI with data based on user actions.

7.1 Data Structures

There are four major objects used in the controller. These are:

- **ListInfo** - This class contains code that exerts overall control over talking to the back end, and stores the information to be displayed on the front end.
- **ListInfoHeading** - Represents a column and its properties.
- **ListInfoItem** - Represents a single item.
- **ListInfoGeneralisation** - Retrieves and puts into a useful form all information on the model to be displayed. This class enables generalisation and thus representation of non classical music data.

7.2 Representing a Displayable Item

ListInfoItem objects describe the information required for a single displayable element. The amount of data required on each individual item is fairly minimal. This includes:

- **URI** - The unique identifier for this item.
- **Label** - The text to display on screen.
- **Column URI** - The unique identifier of the items column. This is used to link back to the column so that any information required on it can easily be obtained.
- **Popularity** - The level of popularity. This field is used for passive power assist to show if the item should be highlighted or not.
- **Sort Item** - The associated item by which this item should be sorted. For example, musical eras might be sorted by their date rather than alphabetically.

7.3 Column Information

ListInfoHeading objects describe a single column. This is largely simple data storage, but also contains methods that describe how to perform queries using this column. Required fields include:

- **URI** - The unique identifier for this column.
- **Label** - The text to display on screen.
- **Type** - The class of objects that are allowed to be displayed in this column. This ensures that queries do not allow erroneous data to be shown on screen.
- **Predicates** - The predicates used to extract what information should be shown in this column.
- **Label Predicate** - The predicate used to extract the text to display on screen for items stored in this column.
- **Selection** - The URI of the selected item in this column.
- **Allowed left/right** - The URIs of columns that are allowed left or right of this column. While there is no code requirement for this item, it may be useful if we wished to disallow certain column layouts as non-useful.
- **Sorting method** - This is an array of ListInfoSortItem objects. These objects describe how to sort items in this column. The default is alphabetical/numerical on the item label, but this enables specifying a different piece of data to sort on. For example, the Era column can be sorted by the eras start or end date.

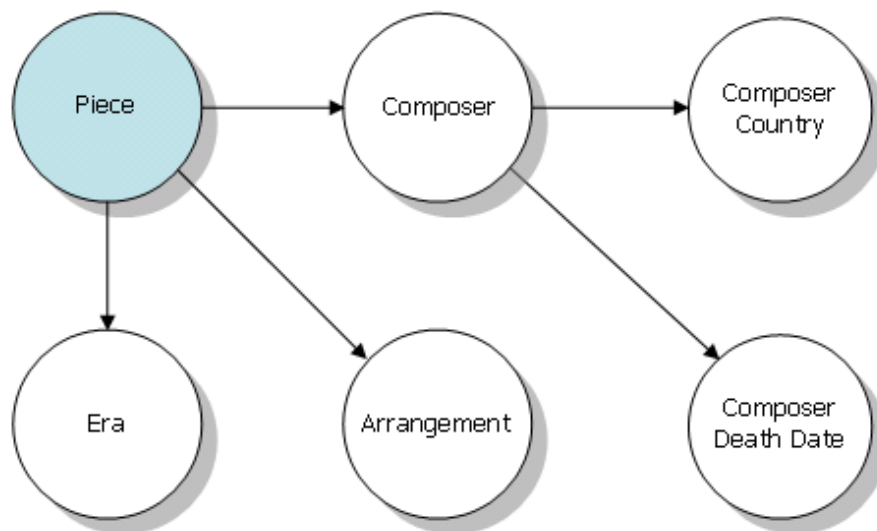


FIGURE 43: RDF Graph

An important concept within this architecture is the goal or object column. This is the column to which all other columns are related. For example, in our classical music data this column is an individual Piece of music. As seen in the graph in Figure 43, each Piece holds data on its composer, arrangement, and era. The predicate for extracting what composers or eras should be in a column are constructed relative to each piece. An example can be seen in Figure 44 below:

```
SELECT ?y WHERE (?x, <mspace:has-era>, ?y) USING mspace FOR
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/>
```

FIGURE 44:

Literally, this RDQL query considers all pieces x , finds their era y , and then returns every unique instance of y . A query that selects all eras with piano arrangements in them would find all pieces with a piano arrangement, and then find every era within that set of pieces. This means that extracting items to fill the goal column with becomes something of a special case; a query to extract all pieces cannot be constructed relative to all pieces. Instead, in the case of a request to list all pieces in the system, a query that asks for all items in the knowledge base that are of the type *Piece* is constructed, as shown in Figure 45. Other implications of this are considered in the Query Formation section.

```
SELECT ?x WHERE (?x, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<mspace:piece>) USING mspace FOR
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/>
```

FIGURE 45:

The *ListInfoHeading* class also provides two important methods. The `getQueryString()` function returns a string representing a query fragment for this column. A simple example is the era column with a selection within it. This function would then return a query fragment that selected all pieces which had the selected eras. This fragment can be seen in Figure 46:

```
(?x, <http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/has-era>,
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/#Baroque-Era>)
```

FIGURE 46:

It should be noted that this fragment is expected to be part of a larger query. To this end, the function allows the specification of the starting variable, the start string for any interim variables used in the fragment, and the end variable (usually the selected item). This allows the fragment to be effectively chained with other query fragments, and prevents variable duplication.

The second major function is `getSortQuery()`. This function is extremely similar to the `getQueryString()` function, and returns a query fragment that retrieving the items each item in this column should be sorted against.

7.4 Populating the Column Array

Our system relies heavily upon querying for populating the UI. There are queries for items such as determining system available columns, updating power assist statistics,

getting preview cues, and many other operations, but these are described in later sections. This section is devoted to the main query method that generates the data to fill the columns.

The `ListInfo` class stores a two dimensional array of `ListInfoItem` objects, representing the column layout in the first dimension and items within the columns in the second. The `ListInfo.update()` method is used to update this array. To ensure reasonable performance, it is possible to specify the range of columns that should be updated.

The first step in this method is to check that the area being updated is actually capable of being updated. If, for example, the column prior to the first column to be updated has no selection, the range of columns will have no data. All columns right of the update area should then be cleared and the method stopped. If this is not the case, the main query loop begins. This loop generates a query to fill a column, runs it, updates the results and runs on the next column. A simple overall outline is shown in Figure 47.

The most complex stage in this process is generating the query. The overall process works by chaining query fragments together. To get the contents of a column, the system chains the predicates of each prior column with the selection in that column, ending with the predicate of the column to be updated and an undefined variable. Finally, type checking is applied and the columns label predicate used to get the displayable text for the item. Figure 48 shows a pseudoquery example of this.

There are several cases to consider when generating the query string:

Case 1 - Updating column 0.

The first column in the array is a special case, because the problem of the object column has to be considered. This column cannot be extracted through a predicate relative to itself, as it is the object of the query. The solution here is to use a query that gets all items that are of the same type as the object column.

Case 2 - Object column present at end.

This is similar to case 1. If we consider Figure ?? in comparison to Figure 48, instead of the last column having a predicate, the variable that is selected is changed to the `x` variable. This returns all the correct object column objects with which to fill the screen.

Case 3 - Object column present but not at end.

This case implies that there is a selection in an object column, and the object column is placed to the left of the column we are updating. In this case the variable `x` that we see in Figure 48 and Figure 49 is already specified. Figure 50 shows an example of this case.

As shown in Figure 47, after this query is generated, there is a section of code that handles grabbing any required sort items. If the column uses the standard sorting

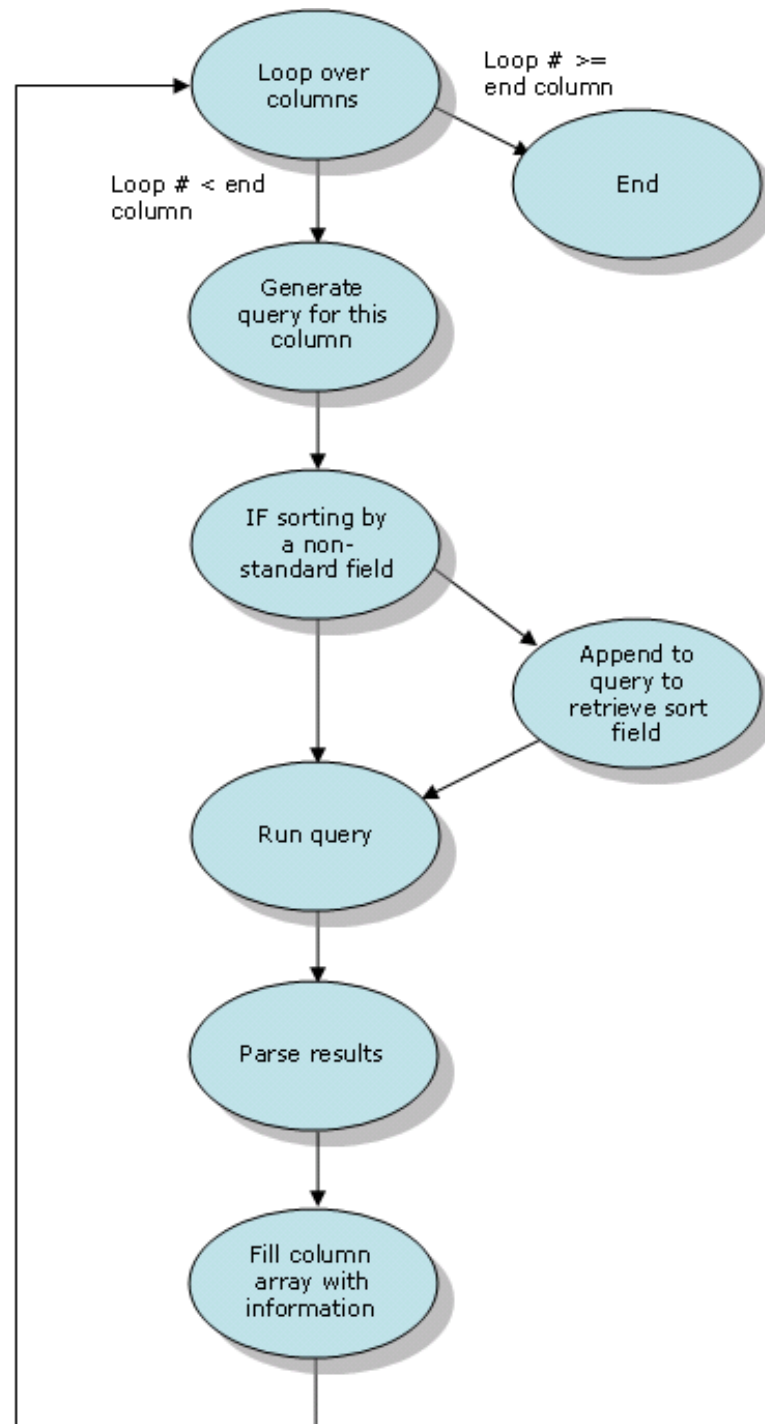


FIGURE 47: Initial Design of the mSpace Browser

SELECT ?y, ?z WHERE (?x, <predicate:column1>, <selection:column1>), (?x, <predicate:column2>, <selection:column2>), (?x, <predicate:column3>, ?y), (?y, <has-type>, <type:column3>), (?y, <labelpredicate:column3>, ?z)

FIGURE 48: Query to update column 3

method (alphabetical/numerical sort on the label), no additional data is required. If it is, the `ListInfoHeading.getSortQuery()` method is called, which provides a query

```
SELECT ?x, ?z WHERE (?x, <predicate:column1>, <selection:column1>), (?x,
<predicate:column2>, <selection:column2>), (?x, <has-type>, <type:column3>), (?x,
<labelpredicate:column3>, ?z)
```

FIGURE 49: Query to update column 3 where column 3 is an object column

```
SELECT ?y, ?z WHERE (<selection:objectcolumn>, <predicate:column3>, ?y), (?y,
<has-type>, <type:column3>), (?y, <labelpredicate:column3>, ?z)
```

FIGURE 50: Query to update column 3 where a prior column is an object column

fragment that can be attached to the end of the main query. It should be noted that the `ListInfoHeading` and `ListInfo` objects support multiple sort methods for each column, although only a single one is currently implemented in the UI. Finally, the query is passed to a backend PHP script that queries the `3Store`, parses the result into XML, and returns to the browser script. This then performs the sort on the results. All results are also added to a hashtable hashed against their URI to allow easy retrieval.

7.4.1 Multi-hop data

As discussed in the background section, RDF is a graph structure. The examples in this section have assumed that the required data is just a single hop away from the goal column. Extracting this data is simple, as everything relates back to the goal column. An added layer of complexity is involved in considering data two or more hops away. A single predicate cannot be used to extract this data. Instead a set of query fragments must be used. An example query to extract all the countries composers have come from is shown in Figure 51, and a pseudoquery showing a 3-hop predicate is shown in Figure 52.

```
SELECT ?y WHERE (?x, <mspace:composed-by>, ?i1), (?i1, <vcard:Country>, ?y)
USING mspace FOR <http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/> vcard
FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

FIGURE 51:

```
SELECT ?y WHERE (?x, <pred1>, ?i1), (?i1, <pred2>, ?i2), (?i3, <pred3>, ?y)
```

FIGURE 52:

Support for using data of an indefinite number of hops away from the goal column is implemented in the system. This allows virtually any piece of RDF data to be meaningfully represented.

7.5 Performing queries - XMLHttpRequest

Most of the work in performing the query once it is formed is handled by the `XMLHttpRequest` object. This object is used to send the query to the backend PHP script

that queries the 3Store, and then parse the returned XML. The XML is rendered into the DOM, allowing the use of standard DOM methods to get the results. In our case, it is a matter of getting an individual result by using the

```
XMLHttpRequest.getElementsByTagName('row')
```

method, and then getting the value of the variables within by examining the `childNodes`.

7.6 Preview Cues

There are three cases for retrieving preview cues for an item.

Case 1 - Preview Goal Item: Each goal item may have zero or more preview items associated with it. If a preview of an item in the goal column is required, the selection is made within that set of preview items. The query in Figure 53 shows how this is retrieved.

```
SELECT ?x, ?z WHERE (<itemuri>,
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/has-preview-filename>, ?x),
(<itemuri>, <labelpredicate>, ?z)
```

FIGURE 53:

Case 2 - Item with assigned previews: Our system provides the opportunity for preview cues to be assigned as good examples of an item, to help the user have as informative an experience as possible. For example, Bachs Air might be considered a good example of a piece from the Baroque era. The query to retrieve data for an item with assigned previews is shown in Figure 54

```
SELECT ?x, ?z WHERE (?y,
<http://mspace.ecs.soton.ac.uk/ontology/is-currently-a-preview-for>, <itemuri>), (?y,
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/has-preview-filename>,
?x), (?y, <labelpredicate>, ?z)
```

FIGURE 54:

Case 3 - item without assigned previews: This case arises when the query described in Figure 54 yields no results. In this case, the system will pick random results that are relevant to the item. The query for this is shown in Figure 55.

```
SELECT ?y, ?z WHERE <column-query-string>, (?x,
<http://mspace.ecs.soton.ac.uk/ontology/ClassicalMusic/has-preview-filename>,
?y), (?x, <labelpredicate>, ?z)
```

FIGURE 55:

When the preview items have been found, an array of their file names and their labels to be displayed on screen is returned.

7.7 Favourites

The ListInfo class stores a list of user-selected favourite items. Favourites can be added by either specifying the URI of the item to add or by passing the item itself. Currently, only items in the goal column are allowed to be added to the favourites list. A useful change in the future would be to distinguish between the goal column and columns whose items can be added to the favourites list. If the add is successful, the method calls for the power assist statistics to be updated and returns.

7.8 Power Assist

Power Assist manifests itself in two forms; an active variant that reorganises the users column layout and selections upon request, and a passive one that simply highlights items that the user may find interesting inside their current layout. Both variants work using the same statistics based off trends in the users favourite selections.

Statistics are updated whenever the user adds a new item to the favourites list. When this happens, the system loops through all of the columns including non-displayed ones, and forms a query for each one to get the data the item is associated with. For example, when a Piece is added to the favourites list, the system would form queries to retrieve its Era, Arrangement, Composer, Composer Country, Composer Death Date, and so on. The URI(s) returned as a result of each of these queries are stored.

The ListInfo class holds two data structures to aid storing statistics: firstly, a hash table with a number hashed against a URI. This stores the number of times each URI has appeared in the information retrieved on a users selections. The second data structure is an array containing a list of the columns. This stores information on the most popular item from each column, what it is and how many times it has been selected. The system loops through all of the URIs returned from the queries and adds/updates them to the hashtable, as well as updating the array of most popular items in each column.

7.8.1 Passive Power Assist

Passive power assist works in a fairly simple manner. It involves the insertion of code into the main query loop where items are getting generated. When each item is created, a method is called to set the items popularity, or the number of times it has been found to be related to an item added to the favourites list. When the items are drawn on screen, if their popularity is above a certain threshold they are highlighted with intensity proportional to the percentage of items in the favourites list they are related to.

7.8.2 Active Power Assist

Active power assist works in a somewhat more complicated fashion. The major difference is in the fact that it has to attempt to select a useful column layout. Ideally, what is required is a large number of columns while still remaining relevant, as this communicates the most information to the user and helps them understand what they enjoy.

The main stage is to loop through the array of the most popular item in each column in order from most popular to least. An initial variable set to 1 is multiplied by `number_of_item_selections/size_of_favourites_list`. Once this value breaches a certain threshold (for example, 0.3), the system breaks out of the loop. The column layout is comprised of items 0 to n-1 in the array of most popular items, where n is the number of loops undertaken before exiting. Finally, the goal column is added to the end of the new column array.

Selections within this new set of columns are handled by passing a parameter to the method that updates the ListInfo objects storage of column contents. This parameter instructs the update method to set the selection to the most popular displayed item in each column. It is impossible to be sure that a highly popular item will be present in any column but the first, thanks to the fact that the selection in the first will restrict the contents of following columns. This means that it cannot be guaranteed that power assist will certainly provide a meaningful answer, although it is likely to. Future work might involve the creation of a better metric for determining column layout to improve the likelihood of popular items occurring in every column.

7.9 Communicating with the User Interface

Interfacing with the UI is handled in the `index.html`. Other classes have no direct communication with the UI, giving a strong degree of separation between the UI and the data that is used to fill it. The `start()` method that is called on page load performs multiple operations: initialising the UI including providing it a surface to draw to, initialising the controller, and loading favourites and column layout (if required) from state. In addition, it provides methods to respond to UI events and provided required information.

7.9.1 Preview Cues

The requirement of the code in `index.html` is simply to respond to the event when the user wishes to view preview cues. Initially, it communicates with the ListInfo object to get the items that are to be used as preview cues. Subsequently, it returns an array of PreviewItems. Each of these contains the URI of a preview cue item, the text to display

for that item, and the html code that will display the cue. Currently this is limited to `<embed src="filename" />`. A useful and simple piece of future work will be to generalise this so that any tag can be used.

7.9.2 Favourites

The controller is responsible for providing the displayable text for favourites list items. This involves a call to the `ListInfo` object, which returns the required data. It looks first at the already displayed objects in case the item being added to the favourites list already exists, in which case the data can be retrieved without a query. If it does not already exist (this case arises when loading favourites from state), it will perform a query to get the required data. Currently this happens on a per item basis. A useful optimisation would be to, when loading from state, request all items at once, allowing the data to be retrieved in a single query.

7.9.3 User initiated events

The `onChange` method is used to handle user events. Since user events can change the state of the system, the current column state is updated for bookmarking on each of these calls. These include:

Click on item: On clicking an item, the `ListInfo` method to repopulate the data is called. The only column that requires repopulation is the one immediately to the right. All columns further to the right are cleared. The UI is then updated with the new data, and the relevant instrumentation method called to log the activity.

Double click on item: Double clicking on an item attempts to add it to the favourites list. Firstly, the system calls the `ListInfo` method `addFavouriteByURI` to attempt to add the item to the internal favourites list. If this succeeds, it adds the item to the UI favourites box, and updates the passive power assist on screen. Lastly, the relevant instrumentation method is called to log the activity.

List move: For the purposes of most changes, whether a column moved left or right is irrelevant; what is important is that two columns were swapped in position. The only data that has to be updated is that in the two columns that are swapped - column contents to the left and right of these remains unchanged. When two columns are swapped it is guaranteed that the currently selected items within will exist in the new data, so selections are preserved and no data is cleared. In response to this event, the system updates the `ListInfo` object with the new positions of the columns, calls for an update on the two moved columns, updates the UI, and subsequently works out whether a column moved left or right for the purposes of calling the correct instrumentation methods.

List removed: The response to list removal is relatively simple. The system tells the ListInfo object to remove a column and update the data, and updates the UI. In this case, all columns to the right of the removed columns must be updated due to their being less limited and thus potentially having extra items to display. Selections are preserved. Lastly, the relevant instrumentation method is called to log the activity.

List added: Again, this is a relatively simple event. A call is made in the ListInfo object to insert the column. All selections to the right of this column are potentially no longer available, so all columns contents and their selections to the right are cleared. The added list is populated based on any selections in the columns to the left. Finally, the relevant instrumentation method is called.

7.9.4 Updating the Screen

Updating the columns on screen is handled in the `assignStuff` method. It has support for only updating certain columns to maximise efficiency. When updating a column, it first clears it, then creates a series of `PreviewListItem` from the data stored in the ListInfo object, which are the visual list items on screen. The method then determines if the item is selected and sets the `setSelected` property if so. Finally, the popularity of the item is determined for use in passive power assist, and the item is added to the list. Once all items have been added, the next column is updated.

7.10 Generalisation of Controller

Generalisation of the controller involves extracting a significant amount of data from the specified mspacemodel. The ListInfo object was designed from the ground up to be receptive to generalisation, in that during the initial classical music-only phase, rather than hard coding column details in multiple areas the column information was provided by stub methods. The generalisation code simply required changing the stub methods to call the correct generalisation method. The queries and the information they are used to extract are discussed below:

Columns

It is impossible to extract all data on a column in a single query. This is due to the fact that few items are *required* to exist. For example, there is no requirement that every column have a predicate that defines how to get the label for items retrieved for that column, because there is a default in place (`<rdfs:label>`). If we attempted to match against multiple optional items, their lack of existence would cause optional items that do exist not to be returned. In future, RDQL will support an OPTIONAL keyword that will make this possible, but for now multiple queries are required.

The only required fields are column predicate, and column label. These are extracted using the query shown in Figure 56.

```
SELECT ?y, ?z, ?z1 WHERE (<modeluri>,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?y), (?y,
    <http://www.w3.org/2000/01/rdf-schema#label>, ?z), (?y,
    <http://mspace.ecs.soton.ac.uk/ontology/#populate-only-with-class>, ?z1)
```

FIGURE 56:

Goal Column

This is a simple query. It simply asks for the model's goal column. The query used is shown in Figure 57.

```
SELECT ?y WHERE (?x,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-goal-column>, ?y), (<" + modeluri + ">,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?y)
```

FIGURE 57:

Default Layout

The mspacemodel requires the definition of a default column layout. This is the layout to present to the user if they aren't loading one from a bookmark or link. The default layout is represented in RDF as a linked list of data, as this enables simple retrieval and ordering. It is possible to extract the entire layout with a single query, as shown in Figure 58.

```
SELECT ?x, ?y, ?z WHERE (?x,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-default-column>, ?z), (?z,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?y), (<modeluri>,
    <http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?y)
```

FIGURE 58:

In this query, x is the previous links URI, y is the URI of the column, and z is the current links URI. Figure 59 helps explain the method by which this works.

In one result of the query, x matches against the model URI, y matches against Column 1, and z matches against Default Link 1. In the other, x matches against Default Link 1, y matches against Column 2, and z matches against Default Link 2. This query structure allows the easiest formation of a linked list in RDF, and is efficient in that all default columns can be retrieved in a single query.

The order that the results are returned in is not guaranteed. For each result, y and z are hashed against x. Since we know that the first value of x will be the URI of the model, once all results are in the system reconstructs the linked list by getting the values hashed against the model URI, then the values hashed against the z value returned, and so on.

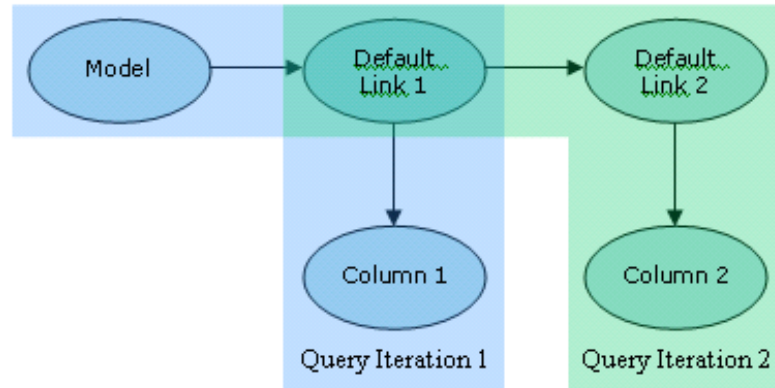


FIGURE 59: Default Column Layout Retrieval

Predicates

Predicate retrieval works in a similar manner to the default layout retrieval, in that predicates are stored in linked lists. The difference in this case is that there is a linked list starting from each column. The required query is shown in Figure 60.

```

SELECT ?x, ?y, ?z WHERE (?x,
<http://mspace.ecs.soton.ac.uk/ontology/#populate-using-predicate>, ?z), (?z,
<http://mspace.ecs.soton.ac.uk/ontology/#has-predicate>, ?y)

```

FIGURE 60:

As with default columns, the results *y* and *z* are hashed against *x*. In this case, there are multiple start points, each one being the URI of a column. Reconstructing the lists involves a loop through each column and loop inside to reconstruct each list. Each list of predicates is then attached to the correct column object.

Item Labels

This is a simple query to get each columns label predicate (the predicate used to get the displayable text on screen for each item in the column). This is an optional field, and defaults to `<rdfs:label>`. The query to retrieve this can be seen in Figure 61.

```

SELECT ?x, ?y WHERE (?x,
<http://mspace.ecs.soton.ac.uk/ontology/#column-label>, ?y), (<modeluri>,
<http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?x)

```

FIGURE 61:

Allowed Right/Left of

This is another fairly simple query. It determines what columns are allowed right of each other, and thus what columns are allowed left of each other. Figure 62 shows the query.

Sort Items

```
SELECT ?x, ?y WHERE (?x,
<http://mspace.ecs.soton.ac.uk/ontology/#allowed-right-of>, ?y), (<modeluri>,
<http://mspace.ecs.soton.ac.uk/ontology/#has-column>, ?x)
```

FIGURE 62:

This is the most complex process currently contained within the generalisation code. Each column can have multiple (ordered) sort predicates, and each of these sort predicates can be multi-hop. Figure 63 shows the full query.

```
SELECT ?x, ?y, ?z2, ?z WHERE (?x,
<http://mspace.ecs.soton.ac.uk/ontology/#sort-method>, ?z), (?z,
<http://mspace.ecs.soton.ac.uk/ontology/#has-predicate-list>, ?y), (?z,
<http://mspace.ecs.soton.ac.uk/ontology/#sort-type>, ?z2)
```

FIGURE 63:

Initially, all sort methods are reconstructed, as shown with other linked list examples. After this, a query is performed for each sort method to return the predicates to access the data to sort against. Note that in this case the data is relative to the item being sorted.

8 Other Implementation

8.1 SourceForge

SourceForge is a popular web site that provides free hosting and web-based collaborative development tools for open source licensed projects[18]. It was always the intention that we mSpace could be used by third parties and we shall be using the free services that SourceForge provide in order to do this. A web site that explains mSpace and how to use it has been created[9].

SourceForge allows users of the software to post bug reports and feature requests. These features of SourceForge have been enabled and we intend to support the project after the initial release in our own time as much as possible. The mailing list feature will allow us to notify users of new releases should they choose to subscribe to this service.

8.2 Populating the Information Box

The data being presented in the information box was discussed in Section 4.1.3. The information for display on the information box is returned to the client via a PHP script which takes a resource identifier as a parameter. This identifier determines which data needs to be returned. The particular information to be displayed is selected from the knowledgebase by use of generalised predicates which specify which predicates to read

from the main resource. For example, each composer's biography is stored using the `has-biography` predicate defined in the mSpace classical music ontology. To indicate that that information should be the main data returned to describe a composer, a triple is created:

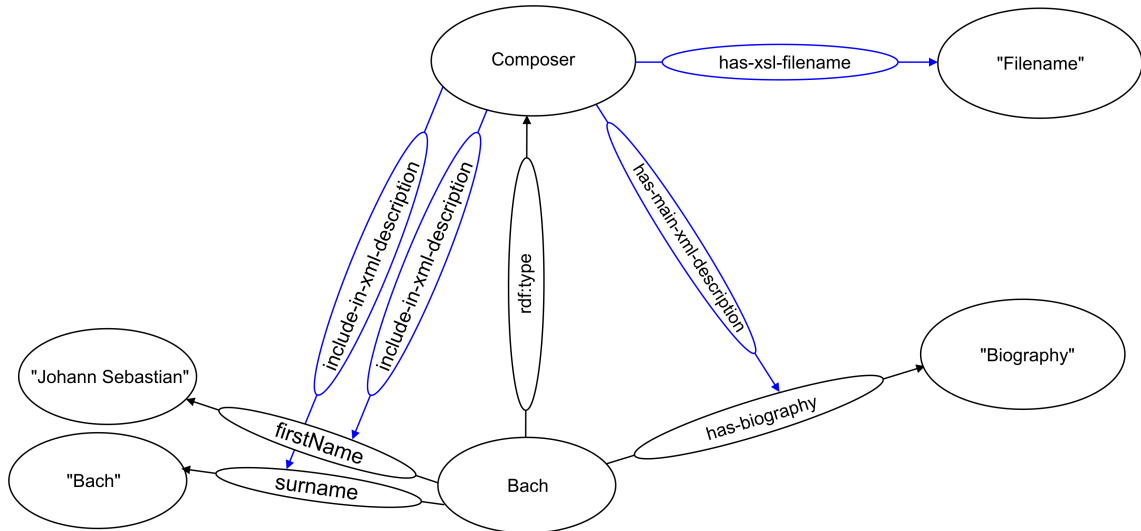


FIGURE 64: The model used to generalise the presentation of data for the information box

There are two types of information which can be used in the information box: some XML as the main content, and additional content. For a composer the additional content could be a birth date or death date. Although the TripleStore accepts XML literals, the data layer between the mSpace scripts and the TripleStore causes the XML to be encoded and thus the script for producing the information box content needs to decode the entities twice.

As multiple data items can be presented in the information box, a generic manner of controlling how these are laid out was desirable. The implemented solution uses XSLT. Each class which could appear in a column can have an XSL filename specified. If the class corresponding to the requested resource has an XSL file specified then the data selected from the TripleStore is put into a new XML document which is then translated server-side into the final content for return to the client. If there is no XSL file available but the main content exists, then the main content is sent directly to the client. Otherwise a suitable search string is produced and the client is sent a link to google with the search string appropriately appended in the href.

Client side the information box is loaded by loading the URL of the `info.php` script as a HTML document. This document is independent of the styles from the main page used for presenting the mSpace user interface.

8.3 Instrumentation

The instrumentation of the mSpace user interface is comprised of a client side module, a server side module and function calls from the main user interface functions to the relevant functions defined in the client side instrumentation module written in JavaScript. The server side module is written in PHP and accepts data from the client side module which is then time stamped, session stamped and written to temporary XML encoded RDF files which are then asserted to the TripleStore using a common model name. The common model name is required so that all instrumentation data can be easily flushed if required. By default the filename (or the URL to the file) would get used and each individual event would have to be independently flushed which is undesirable. Due to the sheer volume of data that instrumentation produces, it is asserted to a separate knowledge base to the core data to avoid any impact on the performance of mSpace exploration.

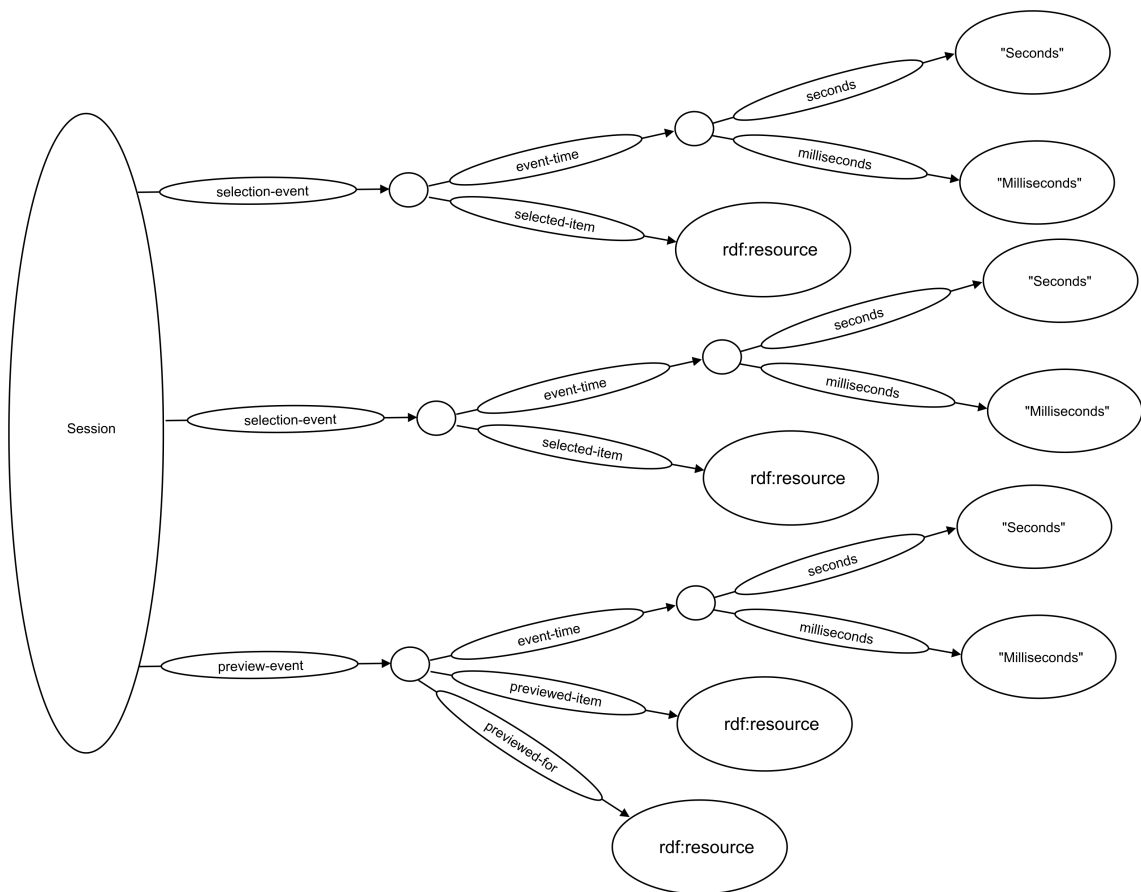


FIGURE 65: The model used for instrumentation logging

The instrumentation modules constructed are focused on the generation and storage of instrumentation data; the statistical processing of such data is not included but would be a very beneficial addition. This will be discussed later in Section 10.3.8. The use of semantics within the instrumentation data is not essential but was chosen due to the semantic nature of the overall solution and also because of the power, flexibility and

compatibility it allows for future processing. Only data directly relevant to each event is stored, as this significantly reduces the volume of data being transmitted, processed and stored; the full state of the user interface can be regenerated by chronological assembly of events which occurred on the session being monitored. The absence of an instrumentation statistics module could impair the utility of the instrumentation, but due to the benefits of semantic logging the instrumentation data can be checked via some simple RDQL queries.

8.4 Goal Count Methodology & Scalability

The goal count calculation script is written in PHP. It is generalised and reads from an mSpace model to determine which classes are being used to populate columns and which classes can have instances which are goal items. RDQL provides no means by which to request a count of items complying to a particular statement / query; it is therefore essential that the computation of such is done in advance and cached due to the excessive time it would take to calculate the goal count on the fly. As the number of goal items appearing beneath a particular item depends on all selections to the left aswell, it is required that every possible combination of selections has a goal count. The nature of RDF and RDQL means that queries will return results which are a superset of what is requested, so the number of selections has to also be stored to greatly simplify the query complexity and execution time.

The number of combinations of instances which could be selected is the product of the number of instances which can be displayed in each column. For example, if there are 5 Eras, 10 composers and 3 arrangements there are $5 \times 10 \times 3 = 150$ combinations, each of these requiring a query and then a result count. The exponential order of complexity means that the implemented goal count algorithm is not very scalable, though it only needs to be executed rarely. A system whereby additions to the knowledgebase automatically make appropriate updates to the goal count cache would significantly improve the scalability and allow it to be used with enormous datasets spanning many dimensions. On the test server it takes approximately 1 hours to compute the goal counts for Era, Arrangement and Composer. Larger and broader datasets will take days to compute goal counts for.

Even with cached results, it still takes time for such to be selected from the TripleStore, this coupled with the fact that the goal counting script doesn't support multi-hop predicates means that it was decided not to implement the display of the goal counts in the client interface.

9 Testing

9.1 Incremental Delivery

Testing is an integral part of any software development process. The solution produced in this project was incrementally delivered which inherently included testing as each stage. The group worked as a team providing critical feedback on each others modules, most feedback led to discussions regarding the most suitable ways to address any issues and to provide the best basis for future progression. Developing in small interdependent units, several of which were done in parallel resulted in a continuous need for integrating and integration testing; all issues arising from incompatibilities in module interfaces were quickly resolved allowing the implementation focus to continually advance.

9.2 User Acceptance Tests

The text of the User Acceptance Tests (UATs) can be found in Appendix C. This section deals with the information drawn from those tests.

9.2.1 Useful Suggestions

There were suggestions in the UATs that stood out. The most obvious item to note is that the performance of the system is below the ideal. This can be improved via shifting the system to a more powerful machine, and by some architecture optimisations such as caching, described further in the Further Work section of our conclusion.

While the system was appreciated as being user friendly, there were suggestions for enhancements: notably making it obvious what items could be added to the Favourites list, and how to add them.

9.2.2 Era Sort

Allowing a column to be sorted by a field other than the label was a suggestion by our supervisor. This allows us to, for example, sort the Era column in the order of the Eras, rather than in the less intuitive alphabetical sorting. Sorting by a non-label column does add somewhat to the amount of data being transferred from server to client, but this is not significantly damaging except in the case of a massive number of results.

9.3 IMDB - Generalisation test

To test the generalisation components of our system, we ran our code on a different data set. We converted the Internet Movie DataBase (IMDB) data into RDF, and created a model file for it. Running the system using the correct model and data gave instantly correct results, as shown in Figure 66. This test shows that we can run our system on non-specified data. It should be noted that currently the system does not run especially effectively in terms of speed, due to the massively larger data set. Possible performance enhancements are discussed in the Future Work section of the conclusion. It should be noted that Figure 66 also shows the back end using OWL features to infer decades from the film year data.

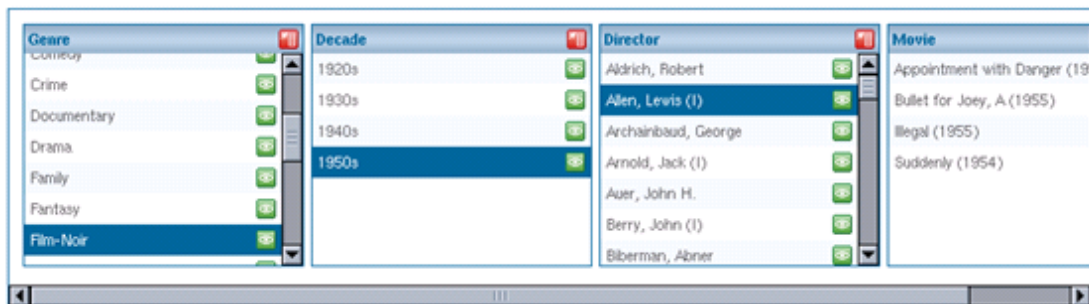


FIGURE 66: RDF Graph

9.4 Test Plans

To ensure that system testing is both relevant and successful, it is important to develop test plans long before they are actually required. The test plans for this mSpace solution can be found in Appendix E. Results of system testing are discussed in the following section.

10 Evaluation

The system (at <http://shaka.ecs.soton.ac.uk/mspace/>) does the specified job well. It meets each of the major criteria in the specification, in that it has:

- UI allowing the user to navigate a column layout and alter that layout.
- Further UI features such as relevance based preview cues, favourites list, and information box.
- Stateful favourites list and saveable column layouts.
- Fully instrumented UI.
- Power Assist feature to help the user understand what they enjoy.

In addition, it implements features beyond the requirements of the specification, including

- Generalisation to allow visualisation of other RDF information.
- Multi-hop data support to further enhance the power generalisation feature.

Testing showed that users found the system helpful, with the major issue being performance rather than usability of the design. Potential improvements to the performance of the system are discussed in the Future Work section of this report.

Our overall conclusion would be that the project was a success. It has delivered a piece of software beyond the limitations of its original spec that will allow the simple creation of semantic web applications, without forcing the use of nonstandard technologies. People wishing to reuse the system are not forced to create time-consuming ontologies for their data if they do not wish to do so, but can leverage the power that they provide if desired.

The project is planning for the future with the creation of a Sourceforge page. This will allow us to continue to improve the system in the future.

10.1 Time Management

In general, most items were completed within the required time period. The specification and User Interface Design were both completed well within their required time periods. In several cases, items were moved where they were dependent upon other items: statefulness, relevance based preview cues, and Power Assist were all dependent upon items within the UI design, which comprised both the UI and the code that populated the UI. This overran slightly, but is much more feature rich than the specification requires.

Due to our commitment to extending the specification through generalisation, the coding of the project as a whole overran by approximately 1 week. Given the benefits realised by this overrun, the group as a whole feels that it was very much worthwhile.

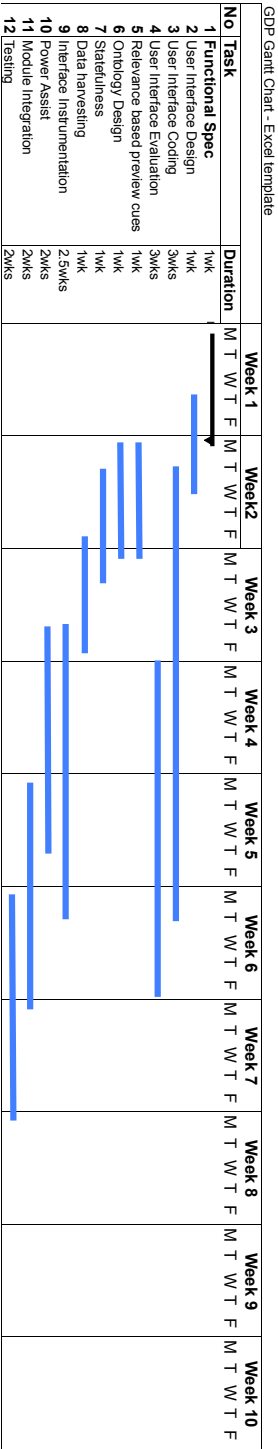


FIGURE 67: Initial Gantt Chart

10.2 User Interface Future Work

10.2.1 UI Component Library

An ongoing task for the mSpace UI is creating a useful component library that can be used to highlight features in other domains of information. This cannot really be classified, but is an ongoing project that will run parallel with any future development of the mSpace project, specifically connecting the browser to different knowledge bases.

10.2.2 Further UI Generalisation

The current UI although generalised already, could also be further generalised.

The current method of message passing relies on the parent object knowing of methods that are required by any child objects. This could be improved by adding the ability to register event handlers. This, although still requiring the parent object to actually know of messages that are going to be passed, means that the methods do not always need to be coded. If the parent does not know about a message then it does not need to provide a method, where as in the current situation a method needs to be provided, even if it is not used, to prevent JavaScript errors.

Another area that could be explored is generalising the UI through the domains knowledge base. A particular item of data within the 3store could specify the type of component that it needs to use, and this information could be passed onto the UI. This would remove the need to alter any of the UI code when changing domains that required different components.

10.2.3 Column/List Filter

A feature that was briefly discussed at the beginning of the project, and even made its way into some of the early designs for the UI, is a Column or List filter. This would be a text box that would appear in the Columns popup menu, and would filter the results in the child List box based on any text the user enters. This is useful when there is many items within a column, but largely only applies to Columns that contain a list.

10.3 Future Work

10.3.1 Scalability

Scalability is an important issue in developing this application further. It can be seen from our IMDB example that the system has issues with scaling to extremely large

datasets and remaining responsive; the script gets stopped by the browser for lack of responsiveness in many cases, and the wait time in general makes it non-useful. There are two major factors controlling this: the time the query takes to run, and the time it takes to parse the query results. Tests have shown the time taken to display the results on screen to be comparatively negligible.

10.3.2 Improving parsing performance

Currently the script uses a significant amount of processing time to parse incoming XML. A possible solution for this is to change the backend script to attempt to return Javascript arrays. This would minimise the required processing time

10.3.3 Improving query times

The creation of a system to cache commonly performed queries has the potential to dramatically increase the speed of the system. While most queries will be performed only irregularly, several will be performed on a much more predictable basis. These include: every query for generalization, greatly increasing the initial load time of the page, and the queries for each possible column when it is the leftmost in the view. Since these are often among the more time consuming queries to perform, system speed should increase noticeably, as well as reducing the load on the server.

10.3.4 Other performance enhancements

There are other methods of improving the responsiveness of the system. A useful way to reduce the amount of time spent querying is to reduce the number of queries. Currently, a significant amount of information gathering has to be separated into multiple queries because not all variables are required to exist, and their lack of existence would cause other information to be lost. A forthcoming addition to RDQL is the `OPTIONAL` keyword. This allows variables to be tagged as not having to be returned if there is no result, and thus allows merging of more queries into one. This would provide a noticeable performance boost due to the reduction of time spent contacting the server.

A further change that might improve perceived response time would be to attempt pseudo-multi-threading. While Javascript does not support a thread model, the `setTimeout()` function does appear to allow the execution of concurrent code. This could be used for functions where the UI ought to remain responsive when running; for example filling the information box or updating power assist statistics.

A simple method for improving performance is to limit the number of results that any query returns using the `LIMIT` keyword. This stops the query and returns the results

once it has reached a certain number of results. This may be necessary when dealing with massive data sets, but potentially stops the user from getting information they may want. A better method is to create column layouts that give only a few results per column; allowing the user to list all films in the IMDB system, for example, would result in having to transfer, parse, and display over 400,000 items, or several megabytes of data.

10.3.5 Streaming

Currently, the system is only accessible within the university network. This is a result of the fact that the media files used for preview cues have to be publicly accessible to allow the browser to retrieve them. Making the media public would infringe upon the university's license to use the files, as it would allow anyone to download them. A useful extension would be to set up a streaming server to allow the media to be viewed but not downloaded.

10.3.6 Multiple selections in a column

Currently the system only allows a single selection in each column. While this is sufficient for most uses, it does limit the ways in which the system can be explored. If the user wishes to explore data in both the Baroque and Romantic eras, or perhaps films in multiple decades, the only way to do so is to select individually each item and explore below it. A useful extension would be to allow multiple selections within columns. This would involve significant changes to the main query loop, and might also have an impact on performance, as it would allow the user to make selections resulting in a large number of results. Further, it would require extensions to the column statefulness code.

10.3.7 Admin tools

Currently, the system is reliant on the model being created in an accurate manner, and to some extent on the RDF data being correctly specified. This is not a major flaw, as the model only has to be created once, but tools to ease creation of the model and error checking on both the model and data would be of use.

Currently, the system attempts to ignore or recover from bad data, through the use of error checking and the ways in which the queries are constructed. The queries prevent bad column names from entering into the data by ensuring that they are specified in the model. A tool that did not implement any such safeguards and caught bad results would be useful to prevent columns getting ignored. Further useful functionality would include checking for bad links in the data, and bad predicate definitions.

A second useful tool would be one to automate the creation of the model RDF. A tool such as this could explore data starting from the user-defined goal column, retrieve predicates and column URIs, and help users create allowed-right-of data and default column layouts. Ideally, this tool would provide a GUI to maximise ease of use.

10.3.8 Instrumentation Analysis

The mSpace interface produced has many features which the development team consider appropriate. It is possible, however, that real world users won't actually use some of them, will find some features difficult, or may prefer alternative interaction mechanisms. We can find out some of this information by speaking to users, but some users may not themselves even know what they want and what will make their user experience better. Analysing the instrumentation data can reveal vast amounts of very useful information regarding the effectiveness of the deployed interaction mechanisms. The timescale for the implementation phase of this project did not allow for additional modules to be written to process the instrumentation data; producing such modules should be one of the next tasks in the progressive development of mSpace as the results may change the way that other future work is designed and how such is integrated with the current version.

The analysis of the instrumentation data will require a user interface. At a minimum a list of sessions should be available, and selecting a session give a human readable chronological log of that session's activities along with some computed statistics regarding how many times they used main features. A further expansion of this would be to use the instrumentation data to recreate and playback a particular session. Computed statistics are an important addition to presentation of logs, tables of results can be output which show which columns are the most popular, which items get selected most often, whether people bother scrolling long columns, how many items get added to favourites, the usage of bookmarked layouts and also the rate of acceptance of power assist suggestions.

The popularity of columns and the reordering of columns can be used to produce a more effective default column layout and this information could also be used by a more advanced power assist system in the future. Knowing which items are selected more often, and whether or not they are previewed prior to selection can help to prove the usefulness of the preview cues and can also be used to update the lists of preview cues by feeding information into the relevant parts of the generalised preview cue mechanism. If an item is regularly selected followed by backtracking then it is likely the user is initially misled into thinking that they would like the item; conversely if after the selection of an item a user continually selects further items and resultantly adds items to favourites then they either already have some knowledge of the domain or the information and preview systems are effectively serving their purpose.

By comparing the selection counts and the preview counts to the positions of items within a column it will be evident whether or not users bother to scroll columns, if they don't then some future work will need to look into ways of better presenting larger volumes of information and ways of ensuring that there is more balanced selection of such. Other than by speaking to the users, the only way to deduce how effective the system is at finding the goal items that they actually want is by analysing the number of items added to favourites and comparing it to the number of steps taken to accomplish such.

Bookmarking a particular set of columns and selections seems to be an obvious benefit both to the current user who may wish to return to the current state, and also to other users with which the current users want to share their view of a knowledge slice. Without analysing the instrumentation data it cannot be confirmed whether bookmarks are a worthwhile feature; the analysis of the bookmark logs needs to compared the number of times slices are recreated from bookmarks to the number of default mSpace layouts issued. The length of time for which a user continues with a session loaded from a bookmark is also of interest.

A Information Box Code Sample

```
/*Method to display a page in the information box*/
function InformationBox_display(url) {

    var divObj = document.getElementById(this.id);
    //If the browser is Gecko-based...
    if (navigator.userAgent.indexOf("Gecko") != -1) {
        //render the data into the DOM
        var obj = document.createElement("object");

        obj.setAttribute("data", url);
        obj.setAttribute("width", (this.width) + this.widthUnit);
        obj.setAttribute("height", (this.height - 21) + this.heightUnit );
        obj.setAttribute("type", "text/html");
        obj.setAttribute("style", "background-color: #FFFFFF");

        while (divObj.hasChildNodes()) {
            divObj.removeChild(divObj.firstChild);
        }
        divObj.style.overflow = "none";
        divObj.appendChild(obj);

    } else {
        //Else, assume IE and set the innerHTML of the object.
        var detail = document.getElementById("detail");
        var detailIE = document.getElementById("detailIE");
        if (!url) {
            url = "./blank.html";
            divObj.style.background = "#FFFFFF";
        } else {
            divObj.style.background = "#EEEECC";
        }
        divObj.style.overflow = "auto";
        divObj.innerHTML=srcDownload(url);
    }
}
```

B 3Store On OSX - HOWTO

3Store on Apple OS X HOWTO

Version 1.0

Authors:

Daniel Smith <das301{at}zepler{dot}net>

History:

Version 1.0 - 2004-11-27

Initial Release

install xcode tools (for c compilation) - these either come with the OS,
or you get them from apple.com

install fink - download from <http://fink.sf.net/>

now run:

```
sudo fink install libgnugetopt
sudo apt-get install mysql mysql12-dev glib
```

install libraptor - download the tarball from
<http://www.redland.opensource.ac.uk/raptor/> (version was 1.4.2 at time
of writing)

untar using stuffit (safari will do this for you, or click "open" in
camino)

now run:

```
cd raptor-1.4.2 (or wherever it is, probably actually
'cd Desktop/raptor-1.4.2')
./configure
make
sudo make install
```

now to compile 3store itself, make sure you're running bash (check by running 'ps', or just type 'bash' if you're not sure)

now run:

```
export CFLAGS="-I/sw/include/gnugetopt -no-cpp-precomp -fno-common  
-I/sw/include/"
```

(this sets up the compiler properly)

at the time of writing, the latest version is 2.2.18, which didn't compile for me (previously 2.2.17 did), however the cvs version did work, so these instructions explain how to download the cvs version and assumes you're using that.

now run:

```
(NB: the following command will ask for the CVS password, it is blank,  
so just hit enter)  
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/threestore login  
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/threestore \  
    co threestore  
cd threestore  
./autogen.sh  
make      (if this fails, run 'sudo make' instead)  
sudo make install
```

presto! 3store is now made, if you want apache support, sadly i've not heard of anybody being successful at compiling the apache module, and i've no experience in debugging apache modules. I did however write a little script in PHP to emulate the behaviour by passing the query to the command like 'tstore_rdql' application and return the results XML-formatted as the apache module does. I DO NOT GUARANTEE THE SECURITY OF THIS SCRIPT - IF IT IS IN USE ON A PUBLIC SYSTEM YOU SHOULD AUDIT THE CODE BEFOREHAND (thanks).

The script is available at <http://www.ecs.soton.ac.uk/~das301/rdql.phpc> also get <http://www.ecs.soton.ac.uk/~das301/config.inc.phpc> - to install this, put in a directory and rename rdql.phpc to index.php, i.e. as /Library/WebServer/Documents/rdql/index.php - rename the config file to config.inc.php and set the config correctly in it (it's easy and the

defaults will be fine too).

To run mysqld, I normally 'sudo su' to be root - then run mysqld_safe, then hit ctrl+z to background, then (because you're running bash, not tcsh) run 'disown'. mysql is now running in the background and killing the terminal will keep it running. refer to the mysql documentation to get this to load on boot and for the proper ways to start it.

To get an empty knowledge base setup, follow the regular 3store instructions that are in the file called README in the threestore directory, there's an automated way (which has never worked for me) and a way that I use under the "manually" section at the end.

C User Acceptance Tests (UATs)

UAT 1

Q. Rate your current knowledge of the classical music domain from 0-10.

A. 4

Q. How would you have previously located data such as this?

A. Google/Other knowledgeable people

Q. What was your initial impression of mSpace, was there anything that stood out?

A. UI is attractively designed.

Q. How easy did you find it to explore classical music with mSpace?

A. Only took a couple of minutes to pick up the feel, generally very useful in terms of finding interesting pieces. I did find that the response times for things like moving columns were a little slow, though.

Q. Did you think the UI (moving, adding columns etc.) was intuitive?

A. Certainly from the point of view of an experienced PC user it was very intuitive. I'd suggest a button on each item in the Pieces category to add an item to favourites, rather than having the double-click thing, though.

Q. Were there any changes you would like to the interface? Please give examples.

A. The one mentioned above, plus I'd like it to be more instantly obvious what the menu button does.

Q. Would you use this system again to explore this domain?

A. Yes.

Q. Would you like to use this interface to browse other domains? Please give

examples.

A. Yes. An IMDB implementation was mentioned which I think would show off the abilities the page has a lot better. Modern music (60s-current) would also work well, showing the very distinct styles available.

Q. Was the information box useful to you? Did you use it?

A. Yes, it provided interesting information. It would be nice to have it available for more selections when possible.

Q. Was the favourites box useful to you? Did you use it?

A. Yes, stored my favourite items as expected.

Q. Do you feel that you now know more about classical music having used mSpace than before? A. Yes, I'd like to spend more time with the web page when possible.

UAT 2

Q. Rate your current knowledge of the classical music domain from 0-10.

A. 6

Q. How would you have previously located data such as this?

A. Web search.

Q. What was your initial impression of mSpace, was there anything that stood out?

A. I had not seen a page much like it before, but it seemed fairly clear where to start.

Q. How easy did you find it to explore classical music with mSpace?

A. Generally each step was fairly obvious. I did find that the Power Assist button provided an interesting layout after I'd been playing with the page for a while.

Q. Did you think the UI (moving, adding columns etc.) was intuitive?

A. Seemed fine.

Q. Were there are changes you would like to the interface? Please give examples.

A. Nothing massive, except it would be nice to be able to keep the preview playing while clicking on other stuff.

Q. Would you use this system again to explore this domain?

A. I generally know how to find information I want on classical music. If I was looking to find out information on more esoteric relationships between bits of data, then certainly.

Q. Would you like to use this interface to browse other domains? Please give examples.

A. Yes. I'm not sure that classical music is the best domain to show off the potential power of the interface.

Q. Was the information box useful to you? Did you use it?

A. Seemed full of useful information, but I'm not too interested in music history.

Q. Was the favourites box useful to you? Did you use it?

A. Yes, and yes. A future extension might include categories in the favourites so that you could have 'playlists'.

Q. Do you feel that you now know more about classical music having used mSpace than before?

A. Yes, somewhat.

UAT 3

Q. Rate your current knowledge of the classical music domain from 0-10.

A. 4

Q. How would you have previously located data such as this?

A. I'm not a fan of classical music and I wouldn't really know where to start looking either, I'd try a search engine.

Q. What was your initial impression of mSpace, was there anything that stood out?

A. Quite plain, not overloaded with links and images as are many websites which provide good amounts of information, I was drawn to selecting an item in the left column and the rest was then quite obvious

Q. How easy did you find it to explore classical music with mSpace?

A. It is quite nice for narrowing down areas of interest by choosing which column is wanted next and then previewing items of interest.

Q. Did you think the UI (moving, adding columns etc.) was intuitive?

A. Seemed fine.

Q. Were there are changes you would like to the interface? Please give examples.

A. It would be nice if the previews stayed in one place.

Q. Would you use this system again to explore this domain?

A. If I ever wanted to find more information about classical music then I would certainly return to mSpace and find out more about what I want before going to an online retailer.

Q. Would you like to use this interface to browse other domains? Please give examples.

A. It would be nice to use the interface for things I am more interested in, such as pop music and movies.

Q. Was the information box useful to you? Did you use it?

A. The information displayed there helped boost my knowledge of composers and eras, it would be nice to have information available for all items, but the google links were also useful.

Q. Was the favourites box useful to you? Did you use it?

A. Yes, the favourites box is good, I used it quite a lot and it was interesting when it offered to rearrange columns based on my choices.

Q. Do you feel that you now know more about classical music having used

mSpace than before?

A. Definitely.

UAT 4

Q. Rate your current knowledge of the classical music domain from 0-10.

A. 5

Q. How would you have previously located data such as this?

A. Browse online retailers or use Google.

Q. What was your initial impression of mSpace, was there anything that stood out?

A. The list of Eras on the left drew my attention.

Q. How easy did you find it to explore classical music with mSpace?

A. Exploring classical music using mSpace was very easy, but it was a bit slow.

Q. Did you think the UI (moving, adding columns etc.) was intuitive?

A. After moving, adding etc once, it became simple to create any layout I wanted to try, the automatic rearranging was useful.

Q. Were there are changes you would like to the interface? Please give examples.

A. I would like the interface to be faster, it would be nice to know how much is left to load while waiting, maybe change column widths to reduce horizontal scrolling.

Q. Would you use this system again to explore this domain?

A. It was nice to be able to see the relationships between the information and to use such to better explore classical music.

Q. Would you like to use this interface to browse other domains? Please give examples.

A. I would certainly like to try it.

Q. Was the information box useful to you? Did you use it?

A. Yes, It was nice to find out more about some things.

Q. Was the favourites box useful to you? Did you use it?

A. The favourites box was good, but it would be better if it could be saved as a separate play-list.

Q. Do you feel that you now know more about classical music having used mSpace than before?

A. Yes.

D Annotated Instrumentation Log Entry

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY owl "http://www.w3.org/2002/07/owl#">
]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:msio=
    "http://mspace.ecs.soton.ac.uk/ontology/instrumentation/">
<rdf:Description about=
  "http://mspace.ecs.soton.ac.uk/sessions
  #5ba59e83ecd972fe6c66244756a50ef01101819489">

  <!-- This "selection-event" logs that a user selected the recorded
        composer at the exact time stored in the event-time-->
<msio:selection-event>

  <msio:event-time>
    <msio:seconds>1103124676</msio:seconds>
    <msio:milliseconds>89127200</msio:milliseconds>
  </msio:event-time>
  <msio:selected-item rdf:resource=
    "http://mspace.ecs.soton.ac.uk/classicalmusic/composers/#83"/>
</msio:selection-event>

  <!-- This "add-favourite-event" record, shows that a user added the
        specified piece to thier favourites-->

<msio:add-favourite-event>
  <msio:event-time>
    <msio:seconds>1103050413</msio:seconds>
    <msio:milliseconds>90327300</msio:milliseconds>
  </msio:event-time>
  <msio:favourite-item rdf:resource=
    "http://mspace.ecs.soton.ac.uk/classicalmusic/pieces/#1493"/>
</msio:add-favourite-event>

</rdf:Description>
</rdf:RDF>
```

E List Of Test Criteria

General

User interface loads without any errors

Pass

Columns

Default column layout loads as specified in model

Pass

All columns have a title

Pass

All columns have a menu button which displays a menu when clicked

Pass

Columns don't scroll when all items are displayable

Pass

Columns scroll to allow viewing of all items when there are too many to display at once

Pass

When too many columns are added to fit the allocated width of column area, a horizontal scrollbar becomes available to allow viewing of all columns

Pass

Columns can be added to the left of other columns

Pass

Columns can be added to the right of other columns

Pass

Columns can be moved right

Pass

Columns can be moved left

Pass

Columns can be closed

Pass

System does not allow closing of all columns

Pass

Status bar informs user that they cannot close last column if they try to do so

Pass

Items

Items in first column load when the interface loads

Pass

Items display with label, appropriate highlight and a preview button

Pass

Items get highlighted when selected

Pass

When an item is selected the next column to the right populates

Pass

When an item is selected the information box populates

Pass

If there is no information to display in information box then an appropriate link for a google search is placed there

Pass

Previews

A preview box appears when a preview button is hovered over

Pass

The preview box lists the available preview cues

Pass

Selecting an item from the list of preview cues causes it to play

Pass

Preview Cue Plugin displays at correct size

Fail

Favourites

When an item in the goal column is double clicked it is added to favourites

Pass

Items in favourites are displayed the same as items in normal columns except for the additional remove button

Pass

Clicking the remove button removes the favourite

Pass

Favourites can be played by using the preview button

Pass

Favourites reload when a user returns to the mSpace interface

Pass

Power Assist

Offers an alternative column layout when there is a noticeable pattern of favourites

Pass

The power assist offer appears in the status bar

Pass

Clicking the status bar when it is displaying a power assist offer will add/remove/rear-range the columns as appropriate

Pass

Never gets enacted when there are less than 5 favourites

Pass

Layout Statefulness

There is a link to a URL containing the state of the column layout

Pass

Clicking the link regenerates the current column layout

Pass

The link can be saved as a bookmark by the user

Pass

A bookmarked link will restore appropriate column layout whenever accessed

Pass

References

- [1] 3store - <http://www.aktors.org/technologies/3store/>.
- [2] Foaf-a-matic - <http://www.ldodds.com/foaf/foaf-a-matic.html>.
- [3] Foaf explorer - <http://xml.mfd-consult.dk/foaf/explorer/>.
- [4] Foaf ontology - <http://xmlns.com/foaf/0.1/>.
- [5] The friend of a friend project (foaf) - <http://www.foaf-project.org/>.
- [6] Google - <http://www.google.com/>.
- [7] Imdb crawling policy - <http://uk.imdb.com/robots.txt>.
- [8] The movie insider - <http://www.themovieinsider.com>.
- [9] mspace - <http://mspace.sf.net>.
- [10] Mysql - <http://www.mysql.com/>.
- [11] Ontology mapping - owl guide -
<http://www.w3.org/tr/2004/rec-owl-guide-20040210/#ontologymapping>.
- [12] Orkut - <http://www.orkut.com/>.
- [13] Owl reference - ifps -
<http://www.w3.org/tr/owl-ref/#inversefunctionalproperty-def>.
- [14] Rdf primer - triples - <http://www.w3.org/tr/rdf-concepts/#dfn-rdf-triple>.
- [15] Rdf site summary / really simple syndication v1.0 specification -
<http://web.resource.org/rss/1.0/>.
- [16] Resource description framework (rdf) / w3c semantic web activity -
<http://www.w3.org/RDF/>.
- [17] Schemaweb -
<http://www.schemaweb.info/>.
- [18] Sourceforge - <http://www.sourceforge.net/>.
- [19] Tribe - <http://www.tribe.net/>.
- [20] University of southampton, department of electronics and computer science - group design project -
<http://www.ecs.soton.ac.uk/ucas/syllabus.php?unit=elec6050>.
- [21] Web ontology language (owl) / w3c semantic web activity -
<http://www.w3.org/2004/OWL/>.

-
- [22] World wide web - size and growth statistics - <http://www.oclc.org/research/projects/archive/wcp/stats/size.htm>.
- [23] The world wide web consortium - <http://www.w3c.org/>.
- [24] Imdb ontology - <http://www.cs.umbc.edu/~skallu1/imdb.pdf>, 2001.
- [25] T. Berners-Lee, J. Hendler, and O. Lasilia. The semantic web - <http://www.scientificamerican.com/article.cfm?articleid=00048144-10d2-1c70-84a9809ec588ef21&catid=2>, 2001.
- [26] L. Besaw. Berkeley unix system calls and interprocess communication - http://www-users.cs.umn.edu/~bentlema/unix/syscalls_and_ipc.html, 1987.
- [27] Apple Computers. Mac os x - <http://www.apple.com/macosx/>.
- [28] Apple Computers. Safari in rss (tiger preview) - <http://www.apple.com/macosx/tiger/safari.html>.
- [29] P. Dawes. Ifpstore - <http://www.phildawes.net/blog/index.php?s=ifpstore>.
- [30] N. Gibbins, S. Harris, A. Dix, and m. schraefel. Applying mspace interfaces to the semantic web - <http://eprints.ecs.soton.ac.uk/10027/>, 2004.
- [31] I. Gilfillan. Database normalization - <http://www.databasejournal.com/sql/etc/article.php/1428511>, 2000.
- [32] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage - <http://eprints.ecs.soton.ac.uk/7970/>, 2003.
- [33] Masahide Kanzaki. Music vocabulary - <http://www.kanzaki.com/ns/music.rdf>.
- [34] Nicole Manktelow. Rise of the blogs - <http://www.smh.com.au/articles/2003/06/06/1054700380653.html>, June 7 2003.
- [35] m. c. schraefel, M. Wilson, and M. Karam. Preview cues: Enhancing access to multimedia content - <http://eprints.ecs.soton.ac.uk/9253/>, 2004.
- [36] A. Seaborne. Rdfql - a query language for rdf - <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>, 2004.
- [37] Network Solutions. Whois search with thumbnails (e.g. microsoft.com) - http://www.networksolutions.com/en_us/whois/, 2004.
- [38] L Westoby. a pedantic web - <http://mms.ecs.soton.ac.uk/mms2003/papers/28.pdf>, 2002.